100个gdb小技巧

wizardforcel

Published with GitBook



目錄

《100个gdb小技巧》

> 作者：hellogcc
>
> 来源：100-gdb-tips

一个关于gdb使用小技巧的文档。100，在这里可能只是表明很多；具体的数目取决于您的参与和贡献。

如何参与

直接发PULL REQUEST，或与我们联系。

增加一个小技巧的步骤：

1. 在src目录下新增一个md文件，参照现有文件的格式风格，编写一个小技巧
   markdown语法参见 http://wowubuntu.com/markdown/
   md文件编写可以使用在线所见即所得编辑器
   https://www.zybuluo.com/mdeditor
2. 在index.md中为新md文件增加一个索引，可以放到已有分类中，或增加一个分类
3. 如果预览下没有问题，OK!

本地生成html的步骤：

1. 确保go和md2min已经安装并可用
2. 直接运行build.sh
3. 如果顺利，会在html目录下生成所有的html文件

联系方式

- 博客网站
- 在线讨论问题：IRC, freenode, #hellogcc房间
- 邮件列表 (发信需要先订阅)

版权

授权许可

致谢

- 各位参与者

其它资源

- GDB在线手册
- GDB命令卡片
- GDB dashboard
- Gdbinit for OS X, iOS and others - x86, x86_64 and ARM
- dotgdb：关于底层调试和反向工程的gdb脚本集

信息显示

显示gdb版本信息

技巧

使用gdb时，如果想查看gdb版本信息，可以使用“`show version`”命令:

```
(gdb) show version
GNU gdb (GDB) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show c
and "show warranty" for details.
This GDB was configured as "x86_64-pc-solaris2.10".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
```

参见gdb手册。

贡献者

nanxiao

显示gdb版权相关信息

技巧

使用gdb时，如果想查看gdb版权相关信息，可以使用“`show copying`”命令:

```
(gdb) show copying
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are desi
to take away your freedom to share and change the works.  By contra
the GNU General Public License is intended to guarantee your freedo
share and change all versions of a program--to make sure it remains
software for all its use
......
```

或者“`show warranty`”命令：

```
(gdb) show warranty
  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRI
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WA
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITE
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICU
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE CO
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRI
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR C
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLU
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT O
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO L
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU O
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGR
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBIL
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provide
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximat
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.
```

参见gdb手册。

贡献者

nanxiao

启动时不显示提示信息

例子

```
$ gdb
GNU gdb (GDB) 7.7.50.20140228-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show c
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
```

技巧

gdb在启动时会显示如上类似的提示信息。

如果不想显示这个信息，则可以使用`-q`选项把提示信息关掉:

```
$ gdb -q
(gdb)
```

你可以在~/.bashrc中，为gdb设置一个别名：

```
alias gdb="gdb -q"
```

详情参见gdb手册

贡献者

xmj

gdb退出时不显示提示信息

技巧

gdb在退出时会提示:

```
A debugging session is active.

    Inferior 1 [process 29686     ] will be killed.

Quit anyway? (y or n) n
```

如果不想显示这个信息，则可以在gdb中使用如下命令把提示信息关掉:

```
(gdb) set confirm off
```

也可以把这个命令加到.gdbinit文件里。

贡献者

nanxiao

输出信息多时不会暂停输出

技巧

有时当gdb输出信息较多时，gdb会暂停输出，并会打印“ `---Type <return> to continue, or q <return> to quit---` ”这样的提示信息，如下面所示：

```
 81 process 2639102      0xff04af84 in __lwp_park () from /usr/lib/
 80 process 2573566      0xff04af84 in __lwp_park () from /usr/lib/
---Type <return> to continue, or q <return> to quit---Quit
```

解决办法是使用“ `set pagination off` ”或者“ `set height 0` ”命令。这样gdb就会全部输出，中间不会暂停。
参见gdb手册.

贡献者

nanxiao

函数

列出函数的名字

例子

```
#include <stdio.h>
#include <pthread.h>
void *thread_func(void *p_arg)
{
        while (1)
        {
                sleep(10);
        }
}
int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread_func, "Thread 1");
        pthread_create(&t2, NULL, thread_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

使用gdb调试时，使用“`info functions`”命令可以列出可执行文件的所有函数名称。以上面代码为例：

```
(gdb) info functions
All defined functions:

File a.c:
int main(void);
void *thread_func(void *);

Non-debugging symbols:
0x0805079c  _PROCEDURE_LINKAGE_TABLE_
0x080507ac  _cleanup@plt
0x080507bc  atexit
0x080507bc  atexit@plt
0x080507cc  __fpstart
0x080507cc  __fpstart@plt
0x080507dc  exit@plt
0x080507ec  __deregister_frame_info_bases@plt
0x080507fc  __register_frame_info_bases@plt
0x0805080c  _Jv_RegisterClasses@plt
0x0805081c  sleep
0x0805081c  sleep@plt
0x0805082c  pthread_create@plt
0x0805083c  _start
0x080508b4  _mcount
0x080508b8  __do_global_dtors_aux
0x08050914  frame_dummy
0x080509f4  __do_global_ctors_aux
0x08050a24  _init
0x08050a31  _fini
```

可以看到会列出函数原型以及不带调试信息的函数。

另外这个命令也支持正则表达式：“`info functions regex`”，这样只会列出符合正则表达式的函数名称，例如：

```
(gdb) info functions thre*
All functions matching regular expression "thre*":

File a.c:
void *thread_func(void *);

Non-debugging symbols:
0x0805082c  pthread_create@plt
```

可以看到gdb只会列出名字里包含“`thre`”的函数。

详情参见gdb手册

贡献者

nanxiao

是否进入带调试信息的函数

例子

```
#include <stdio.h>

int func(void)
{
    return 3;
}

int main(void)
{
    int a = 0;

    a = func();
    printf("%d\n", a);
    return 0;
}
```

技巧

使用gdb调试遇到函数时，使用step命令（缩写为s）可以进入函数（函数必须有调试信息）。以上面代码为例：

```
(gdb) n
12              a = func();
(gdb) s
func () at a.c:5
5               return 3;
(gdb) n
6       }
(gdb)
main () at a.c:13
13              printf("%d\n", a);
```

可以看到gdb进入了func函数。

可以使用next命令（缩写为n）不进入函数，gdb会等函数执行完，再显示下一行要执行的程序代码：

```
(gdb) n
12                a = func();
(gdb) n
13                printf("%d\n", a);
(gdb) n
3
14                return 0;
```

可以看到gdb没有进入func函数。

详情参见gdb手册

贡献者

nanxiao

进入不带调试信息的函数

例子

```
#include <stdio.h>
#include <pthread.h>

typedef struct
{
        int a;
        int b;
        int c;
        int d;
        pthread_mutex_t mutex;
}ex_st;

int main(void) {
        ex_st st = {1, 2, 3, 4, PTHREAD_MUTEX_INITIALIZER};
        printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
        return 0;
}
```

技巧

默认情况下，gdb不会进入不带调试信息的函数。以上面代码为例：

```
(gdb) n
15              printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
(gdb) s
1,2,3,4
16              return 0;
```

可以看到由于printf函数不带调试信息，所以“s”命令（s是“step”缩写）无法进入printf函数。

可以执行“set step-mode on”命令，这样gdb就不会跳过没有调试信息的函数：

```
(gdb) set step-mode on
(gdb) n
15              printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
(gdb) s
0x00007ffff7a993b0 in printf () from /lib64/libc.so.6
(gdb) s
0x00007ffff7a993b7 in printf () from /lib64/libc.so.6
```

可以看到gdb进入了printf函数，接下来可以使用调试汇编程序的办法去调试函数。

详情参见gdb手册

贡献者

nanxiao

退出正在调试的函数

例子

```
#include <stdio.h>

int func(void)
{
    int i = 0;

    i += 2;
    i *= 10;

    return i;
}

int main(void)
{
    int a = 0;

    a = func();
    printf("%d\n", a);
    return 0;
}
```

技巧

当单步调试一个函数时，如果不想继续跟踪下去了，可以有两种方式退出。

第一种用“`finish`”命令，这样函数会继续执行完，并且打印返回值，然后等待输入接下来的命令。以上面代码为例：

```
(gdb) n
17            a = func();
(gdb) s
func () at a.c:5
5                 int i = 0;
(gdb) n
7                 i += 2;
(gdb) fin
find    finish
(gdb) finish
Run till exit from #0  func () at a.c:7
0x08050978 in main () at a.c:17
17            a = func();
Value returned is $1 = 20
```

可以看到当不想再继续跟踪`func`函数时，执行完“`finish`”命令，gdb会打印结果：“`20`”，然后停在那里。

详情参见gdb手册

第二种用“`return`”命令，这样函数不会继续执行下面的语句，而是直接返回。也可以用“`return expression`”命令指定函数的返回值。仍以上面代码为例：

```
(gdb) n
17            a = func();
(gdb) s
func () at a.c:5
5                 int i = 0;
(gdb) n
7                 i += 2;
(gdb) n
8                 i *= 10;
(gdb) re
record              remove-inferiors    return              reverse
refresh             remove-symbol-file  reverse-continue    reverse
remote              restore             reverse-finish      reverse
(gdb) return 40
Make func return now? (y or n) y
#0  0x08050978 in main () at a.c:17
17            a = func();
(gdb) n
18            printf("%d\n", a);
(gdb)
40
19            return 0;
```

可以看到“`return`”命令退出了函数并且修改了函数的返回值。

详情参见gdb手册

贡献者

nanxiao

直接执行函数

例子

```
#include <stdio.h>

int global = 1;

int func(void)
{
    return (++global);
}

int main(void)
{
    printf("%d\n", global);
    return 0;
}
```

技巧

使用gdb调试程序时，可以使用“`call`”或“`print`”命令直接调用函数执行。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x4004e3: file a.c, line 12.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:12
12                printf("%d\n", global);
(gdb) call func()
$1 = 2
(gdb) print func()
$2 = 3
(gdb) n
3
13                return 0;
```

可以看到执行两次`func`函数后，`global`的值变成`3`。
参见gdb手册.

贡献者

nanxiao

打印函数堆栈帧信息

例子

```
#include <stdio.h>
int func(int a, int b)
{
    int c = a * b;
    printf("c is %d\n", c);
}

int main(void)
{
    func(1, 2);
    return 0;
}
```

技巧

使用gdb调试程序时，可以使用“`i frame`”命令（`i`是`info`命令缩写）显示函数堆栈帧信息。以上面程序为例：

```
Breakpoint 1, func (a=1, b=2) at a.c:5
5                printf("c is %d\n", c);
(gdb) i frame
Stack level 0, frame at 0x7fffffffe590:
 rip = 0x40054e in func (a.c:5); saved rip = 0x400577
 called by frame at 0x7fffffffe5a0
 source language c.
 Arglist at 0x7fffffffe580, args: a=1, b=2
 Locals at 0x7fffffffe580, Previous frame's sp is 0x7fffffffe590
 Saved registers:
  rbp at 0x7fffffffe580, rip at 0x7fffffffe588
(gdb) i registers
rax            0x2      2
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe688   140737488348808
rsi            0x2      2
rdi            0x1      1
rbp            0x7fffffffe580   0x7fffffffe580
rsp            0x7fffffffe560   0x7fffffffe560
r8             0x7ffff7dd4e80   140737351863936
r9             0x7ffff7dea560   140737351951712
r10            0x7fffffffe420   140737488348192
```

```
r11            0x7ffff7a35dd0   140737348066768
r12            0x400440 4195392
r13            0x7fffffffe670   140737488348784
r14            0x0      0
r15            0x0      0
rip            0x40054e 0x40054e <func+24>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) disassemble func
Dump of assembler code for function func:
   0x0000000000400536 <+0>:     push   %rbp
   0x0000000000400537 <+1>:     mov    %rsp,%rbp
   0x000000000040053a <+4>:     sub    $0x20,%rsp
   0x000000000040053e <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000400541 <+11>:    mov    %esi,-0x18(%rbp)
   0x0000000000400544 <+14>:    mov    -0x14(%rbp),%eax
   0x0000000000400547 <+17>:    imul   -0x18(%rbp),%eax
   0x000000000040054b <+21>:    mov    %eax,-0x4(%rbp)
=> 0x000000000040054e <+24>:    mov    -0x4(%rbp),%eax
   0x0000000000400551 <+27>:    mov    %eax,%esi
   0x0000000000400553 <+29>:    mov    $0x400604,%edi
   0x0000000000400558 <+34>:    mov    $0x0,%eax
   0x000000000040055d <+39>:    callq  0x400410 <printf@plt>
   0x0000000000400562 <+44>:    leaveq
   0x0000000000400563 <+45>:    retq
End of assembler dump.
```

可以看到执行“`i frame`”命令后，输出了当前函数堆栈帧的地址，指令寄存器的值，局部变量地址及值等信息，可以对照当前寄存器的值和函数的汇编指令看一下。

参见gdb手册.

贡献者

nanxiao

打印尾调用堆栈帧信息

例子

```
#include<stdio.h>
void a(void)
{
        printf("Tail call frame\n");
}

void b(void)
{
        a();
}

void c(void)
{
        b();
}

int main(void)
{
        c();
        return 0;
}
```

技巧

当一个函数最后一条指令是调用另外一个函数时，开启优化选项的编译器常常以最后被调用的函数返回值作为调用者的返回值，这称之为“尾调用（Tail call）”。以上面程序为例，编译程序（使用‘-O’）：

```
gcc -g -O -o test test.c
```

查看 `main` 函数汇编代码：

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000400565 <+0>:     sub    $0x8,%rsp
0x0000000000400569 <+4>:     callq  0x400536 <a>
0x000000000040056e <+9>:     mov    $0x0,%eax
0x0000000000400573 <+14>:    add    $0x8,%rsp
0x0000000000400577 <+18>:    retq
```

可以看到`main`函数直接调用了函数`a`，根本看不到函数`b`和函数`c`的影子。

在函数`a`入口处打上断点，程序停止后，打印堆栈帧信息：

```
(gdb) i frame
Stack level 0, frame at 0x7fffffffe590:
 rip = 0x400536 in a (test.c:4); saved rip = 0x40056e
 called by frame at 0x7fffffffe5a0
 source language c.
 Arglist at 0x7fffffffe580, args:
 Locals at 0x7fffffffe580, Previous frame's sp is 0x7fffffffe590
 Saved registers:
  rip at 0x7fffffffe588
```

看不到尾调用的相关信息。

可以设置“`debug entry-values`”选项为非0的值，这样除了输出正常的函数堆栈帧信息以外，还可以输出尾调用的相关信息：

```
(gdb) set debug entry-values 1
(gdb) b test.c:4
Breakpoint 1 at 0x400536: file test.c, line 4.
(gdb) r
Starting program: /home/nanxiao/test

Breakpoint 1, a () at test.c:4
4       {
(gdb) i frame
tailcall: initial:
Stack level 0, frame at 0x7fffffffe590:
 rip = 0x400536 in a (test.c:4); saved rip = 0x40056e
 called by frame at 0x7fffffffe5a0
 source language c.
 Arglist at 0x7fffffffe580, args:
 Locals at 0x7fffffffe580, Previous frame's sp is 0x7fffffffe590
 Saved registers:
  rip at 0x7fffffffe588
```

可以看到输出了“`tailcall: initial:`”信息。

参见gdb手册.

贡献者

nanxiao

选择函数堆栈帧

例子

```
#include <stdio.h>

int func1(int a)
{
        return 2 * a;
}

int func2(int a)
{
        int c = 0;
        c = 2 * func1(a);
        return c;
}

int func3(int a)
{
        int c = 0;
        c = 2 * func2(a);
        return c;
}

int main(void)
{
        printf("%d\n", func3(10));
        return 0;
}
```

技巧

用gdb调试程序时，当程序暂停后，可以用“`frame n`”命令选择函数堆栈帧，其中`n`是层数。以上面程序为例：

```
(gdb) b test.c:5
Breakpoint 1 at 0x40053d: file test.c, line 5.
(gdb) r
Starting program: /home/nanxiao/test

Breakpoint 1, func1 (a=10) at test.c:5
5               return 2 * a;
(gdb) bt
#0  func1 (a=10) at test.c:5
#1  0x0000000000400560 in func2 (a=10) at test.c:11
#2  0x0000000000400586 in func3 (a=10) at test.c:18
#3  0x000000000040059e in main () at test.c:24
(gdb) frame 2
#2  0x0000000000400586 in func3 (a=10) at test.c:18
18              c = 2 * func2(a);
```

可以看到程序断住后，最内层的函数帧为第 `0` 帧。执行 `frame 2` 命令后，当前的堆栈帧变成了 `fun3` 的函数帧。

也可以用“ `frame addr` ”命令选择函数堆栈帧，其中 `addr` 是堆栈地址。仍以上面程序为例：

```
(gdb) frame 2
#2  0x0000000000400586 in func3 (a=10) at test.c:18
18              c = 2 * func2(a);
(gdb) i frame
Stack level 2, frame at 0x7fffffffe590:
 rip = 0x400586 in func3 (test.c:18); saved rip = 0x40059e
 called by frame at 0x7fffffffe5a0, caller of frame at 0x7fffffffe5
 source language c.
 Arglist at 0x7fffffffe580, args: a=10
 Locals at 0x7fffffffe580, Previous frame's sp is 0x7fffffffe590
 Saved registers:
  rbp at 0x7fffffffe580, rip at 0x7fffffffe588
(gdb) frame 0x7fffffffe568
#1  0x0000000000400560 in func2 (a=10) at test.c:11
11              c = 2 * func1(a);
```

使用“ `i frame` ”命令可以知道 `0x7fffffffe568` 是 `func2` 的函数堆栈帧地址，使用“ `frame 0x7fffffffe568` ”可以切换到 `func2` 的函数堆栈帧。

参见gdb手册.

贡献者

nanxiao

向上或向下切换函数堆栈帧

例子

```
#include <stdio.h>

int func1(int a)
{
        return 2 * a;
}

int func2(int a)
{
        int c = 0;
        c = 2 * func1(a);
        return c;
}

int func3(int a)
{
        int c = 0;
        c = 2 * func2(a);
        return c;
}

int main(void)
{
        printf("%d\n", func3(10));
        return 0;
}
```

技巧

用gdb调试程序时，当程序暂停后，可以用“`up n`”或“`down n`”命令向上或向下选择函数堆栈帧，其中`n`是层数。以上面程序为例：

```
(gdb) b test.c:5
Breakpoint 1 at 0x40053d: file test.c, line 5.
(gdb) r
Starting program: /home/nanxiao/test

Breakpoint 1, func1 (a=10) at test.c:5
5               return 2 * a;
(gdb) bt
#0  func1 (a=10) at test.c:5
#1  0x0000000000400560 in func2 (a=10) at test.c:11
#2  0x0000000000400586 in func3 (a=10) at test.c:18
#3  0x000000000040059e in main () at test.c:24
(gdb) frame 2
#2  0x0000000000400586 in func3 (a=10) at test.c:18
18              c = 2 * func2(a);
(gdb) up 1
#3  0x000000000040059e in main () at test.c:24
24              printf("%d\n", func3(10));
(gdb) down 2
#1  0x0000000000400560 in func2 (a=10) at test.c:11
11              c = 2 * func1(a);
```

可以看到程序断住后，先执行“`frame 2`”命令，切换到`fun3`函数。接着执行“`up 1`”命令，此时会切换到`main`函数，也就是会往外层的堆栈帧移动一层。反之，当执行“`down 2`”命令后，又会向内层堆栈帧移动二层。如果不指定`n`，则`n`默认为`1`.

还有“`up-silently n`”和“`down-silently n`”这两个命令，与“`up n`”和“`down n`”命令区别在于，切换堆栈帧后，不会打印信息，仍以上面程序为例：

```
(gdb) up
#2  0x0000000000400586 in func3 (a=10) at test.c:18
18              c = 2 * func2(a);
(gdb) bt
#0  func1 (a=10) at test.c:5
#1  0x0000000000400560 in func2 (a=10) at test.c:11
#2  0x0000000000400586 in func3 (a=10) at test.c:18
#3  0x000000000040059e in main () at test.c:24
(gdb) up-silently
(gdb) i frame
Stack level 3, frame at 0x7fffffffe5a0:
 rip = 0x40059e in main (test.c:24); saved rip = 0x7ffff7a35ec5
 caller of frame at 0x7fffffffe590
 source language c.
 Arglist at 0x7fffffffe590, args:
 Locals at 0x7fffffffe590, Previous frame's sp is 0x7fffffffe5a0
 Saved registers:
  rbp at 0x7fffffffe590, rip at 0x7fffffffe598
```

可以看到从 `func3` 切换到 `main` 函数堆栈帧时，并没有打印出相关信息。

参见gdb手册.

贡献者

nanxiao

断点

在匿名空间设置断点

例子

```
namespace Foo
{
  void foo()
  {
  }
}

namespace
{
  void bar()
  {
  }
}
```

技巧

在gdb中，如果要对namespace Foo中的foo函数设置断点，可以使用如下命令：

```
(gdb) b Foo::foo
```

如果要对匿名空间中的bar函数设置断点，可以使用如下命令：

```
(gdb) b (anonymous namespace)::bar
```

贡献者

xmj

在程序地址上打断点

例子

```
0000000000400522 <main>:
  400522:       55                      push   %rbp
  400523:       48 89 e5                mov    %rsp,%rbp
  400526:       8b 05 00 1b 00 00       mov    0x1b00(%rip),%eax
  40052c:       85 c0                   test   %eax,%eax
  40052e:       75 07                   jne    400537 <main+0x15>
  400530:       b8 7c 06 40 00          mov    $0x40067c,%eax
  400535:       eb 05                   jmp    40053c <main+0x1a>
```

技巧

当调试汇编程序，或者没有调试信息的程序时，经常需要在程序地址上打断点，方法为 `b *address` 。例如：

```
(gdb) b *0x400522
```

详情参见gdb手册

贡献者

xmj

在程序入口处打断点

获取程序入口

方法一：

```
$ strip a.out
$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400440
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4496 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

方法二：

```
$ gdb a.out
>>> info files
Symbols from "/home/me/a.out".
Local exec file:
    `/home/me/a.out', file type elf64-x86-64.
    Entry point: 0x400440
    0x0000000000400238 - 0x0000000000400254 is .interp
    0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
    0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
    0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
    0x00000000004002b8 - 0x0000000000400318 is .dynsym
    0x0000000000400318 - 0x0000000000400355 is .dynstr
    0x0000000000400356 - 0x000000000040035e is .gnu.version
    0x0000000000400360 - 0x0000000000400380 is .gnu.version_r
    0x0000000000400380 - 0x0000000000400398 is .rela.dyn
    0x0000000000400398 - 0x00000000004003e0 is .rela.plt
    0x00000000004003e0 - 0x00000000004003fa is .init
    0x0000000000400400 - 0x0000000000400440 is .plt
    0x0000000000400440 - 0x00000000004005c2 is .text
    0x00000000004005c4 - 0x00000000004005cd is .fini
    0x00000000004005d0 - 0x00000000004005e0 is .rodata
    0x00000000004005e0 - 0x0000000000400614 is .eh_frame_hdr
    0x0000000000400618 - 0x000000000040070c is .eh_frame
    0x0000000000600e10 - 0x0000000000600e18 is .init_array
    0x0000000000600e18 - 0x0000000000600e20 is .fini_array
    0x0000000000600e20 - 0x0000000000600e28 is .jcr
    0x0000000000600e28 - 0x0000000000600ff8 is .dynamic
    0x0000000000600ff8 - 0x0000000000601000 is .got
    0x0000000000601000 - 0x0000000000601030 is .got.plt
    0x0000000000601030 - 0x0000000000601040 is .data
    0x0000000000601040 - 0x0000000000601048 is .bss
```

技巧

当调试没有调试信息的程序时，直接运行 `start` 命令是没有效果的：

```
(gdb) start
Function "main" not defined.
```

如果不知道main在何处，那么可以在程序入口处打断点。先通过 `readelf` 或者进入gdb，执行 `info files` 获得入口地址，然后：

```
(gdb) b *0x400440
(gdb) r
```

贡献者

* xmj
* weekface

在文件行号上打断点

例子

```
/* a/file.c */
#include <stdio.h>

void print_a (void)
{
  puts ("a");
}

/* b/file.c */
#include <stdio.h>

void print_b (void)
{
  puts ("b");
}

/* main.c */
extern void print_a(void);
extern void print_b(void);

int main(void)
{
  print_a();
  print_b();
  return 0;
}
```

技巧

这个比较简单，如果要在当前文件中的某一行打断点，直接`b linenum`即可，例如：

```
(gdb) b 7
```

也可以显式指定文件，`b file:linenum`例如：

```
(gdb) b file.c:6
Breakpoint 1 at 0x40053b: file.c:6. (2 locations)
(gdb) i breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   <MULTIPLE>
1.1                         y     0x000000000040053b in print_a at
1.2                         y     0x000000000040054b in print_b at
```

可以看出，gdb会对所有匹配的文件设置断点。你可以通过指定（部分）路径，来区分相同的文件名：

```
(gdb) b a/file.c:6
```

注意：通过行号进行设置断点的一个弊端是，如果你更改了源程序，那么之前设置的断点就可能不是你想要的了。

详情参见gdb手册

贡献者

xmj

保存已经设置的断点

例子

```
$ gdb -q `which gdb`
Reading symbols from /home/xmj/install/binutils-trunk/bin/gdb...dor
(gdb) b gdb_main
Breakpoint 1 at 0x5a7af0: file /home/xmj/project/binutils-trunk/gdb
(gdb) b captured_main
Breakpoint 2 at 0x5a6bd0: file /home/xmj/project/binutils-trunk/gdb
(gdb) b captured_command_loop
Breakpoint 3 at 0x5a68b0: file /home/xmj/project/binutils-trunk/gdb
```

技巧

在gdb中，可以使用如下命令将设置的断点保存下来：

```
(gdb) save breakpoints file-name-to-save
```

下此调试时，可以使用如下命令批量设置保存的断点：

```
(gdb) source file-name-to-save
```

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000005a7af0 in gdb_main at /
2       breakpoint     keep y   0x00000000005a6bd0 in captured_main
3       breakpoint     keep y   0x00000000005a68b0 in captured_comm
```

详情参见gdb手册

贡献者

xmj

设置临时断点

例子

```
#include <stdio.h>
#include <pthread.h>

typedef struct
{
        int a;
        int b;
        int c;
        int d;
        pthread_mutex_t mutex;
}ex_st;

int main(void) {
        ex_st st = {1, 2, 3, 4, PTHREAD_MUTEX_INITIALIZER};
        printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
        return 0;
}
```

技巧

在使用gdb时，如果想让断点只生效一次，可以使用“tbreak”命令（缩写为：tb）。以上面程序为例：

```
(gdb) tb a.c:15
Temporary breakpoint 1 at 0x400500: file a.c, line 15.
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     del  y   0x0000000000400500 in main at a.c:1
(gdb) r
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:15
15              printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
(gdb) i b
No breakpoints or watchpoints.
```

首先在文件的第15行设置临时断点，当程序断住后，用“i b”（"info breakpoints"缩写）命令查看断点，发现断点没有了。也就是断点命中一次后，就被删掉了。

详情参见gdb手册

贡献者

nanxiao

设置条件断点

例子

```
#include <stdio.h>

int main(void)
{
        int i = 0;
        int sum = 0;

        for (i = 1; i <= 200; i++)
        {
            sum += i;
        }

        printf("%d\n", sum);
        return 0;
}
```

技巧

gdb可以设置条件断点，也就是只有在条件满足时，断点才会被触发，命令是“`break … if cond`”。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x4004cc: file a.c, line 5.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:5
5                       int i = 0;
(gdb) b 10 if i==101
Breakpoint 2 at 0x4004e3: file a.c, line 10.
(gdb) r
Starting program: /data2/home/nanxiao/a

Breakpoint 2, main () at a.c:10
10                                      sum += i;
(gdb) p sum
$1 = 5050
```

可以看到设定断点只在`i`的值为`101`时触发，此时打印`sum`的值为`5050`。

详情参见gdb手册

贡献者

nanxiao

忽略断点

例子

```
#include <stdio.h>

int main(void)
{
        int i = 0;
        int sum = 0;

        for (i = 1; i <= 200; i++)
        {
            sum += i;
        }

        printf("%d\n", sum);
        return 0;
}
```

技巧

在设置断点以后，可以忽略断点，命令是“`ignore bnum count`”：意思是接下来`count`次编号为`bnum`的断点触发都不会让程序中断，只有第`count + 1`次断点触发才会让程序中断。以上面程序为例：

```
(gdb) b 10
Breakpoint 1 at 0x4004e3: file a.c, line 10.
(gdb) ignore 1 5
Will ignore next 5 crossings of breakpoint 1.
(gdb) r
Starting program: /data2/home/nanxiao/a

Breakpoint 1, main () at a.c:10
10                                      sum += i;
(gdb) p i
$1 = 6
```

可以看到设定忽略断点前`5`次触发后，第一次断点断住时，打印`i`的值是`6`。如果想让断点下次就生效，可以将`count`置为`0`：“`ignore 1 0`”。

详情参见gdb手册

贡献者

nanxiao

观察点

设置观察点

例子

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
int a = 0;

void *thread1_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

int main(int argc, char* argv[])
{
        pthread_t t1;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");

        sleep(1000);
        return 0;
}
```

技巧

gdb可以使用“`watch`”命令设置观察点，也就是当一个变量值发生变化时，程序会停下来。以上面程序为例:

```
(gdb) start
Temporary breakpoint 1 at 0x4005a8: file a.c, line 19.
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Temporary breakpoint 1, main () at a.c:19
19              pthread_create(&t1, NULL, thread1_func, "Thread 1")
(gdb) watch a
Hardware watchpoint 2: a
(gdb) r
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff782c700 (LWP 8813)]
[Switching to Thread 0x7ffff782c700 (LWP 8813)]
Hardware watchpoint 2: a

Old value = 0
New value = 1
thread1_func (p_arg=0x4006d8) at a.c:11
11                      sleep(10);
(gdb) c
Continuing.
Hardware watchpoint 2: a

Old value = 1
New value = 2
thread1_func (p_arg=0x4006d8) at a.c:11
11                      sleep(10);
```

可以看到，使用“`watch a`”命令以后，当`a`的值变化：由`0`变成`1`，由`1`变成`2`，程序都会停下来。
此外也可以使用“`watch *(data type*)address`”这样的命令，仍以上面程序为例:

```
(gdb) p &a
$1 = (int *) 0x6009c8 <a>
(gdb) watch *(int*)0x6009c8
Hardware watchpoint 2: *(int*)0x6009c8
(gdb) r
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff782c700 (LWP 15431)]
[Switching to Thread 0x7ffff782c700 (LWP 15431)]
Hardware watchpoint 2: *(int*)0x6009c8

Old value = 0
New value = 1
thread1_func (p_arg=0x4006d8) at a.c:11
11                        sleep(10);
(gdb) c
Continuing.
Hardware watchpoint 2: *(int*)0x6009c8

Old value = 1
New value = 2
thread1_func (p_arg=0x4006d8) at a.c:11
11                        sleep(10);
```

先得到`a`的地址：`0x6009c8`，接着用“`watch *(int*)0x6009c8`”设置观察点，可以看到同“`watch a`”命令效果一样。
观察点可以通过软件或硬件的方式实现，取决于具体的系统。但是软件实现的观察点会导致程序运行很慢，使用时需注意。参见gdb手册.

如果系统支持硬件观测的话，当设置观测点是会打印如下信息： Hardware watchpoint num: expr

如果不想用硬件观测点的话可如下设置： set can-use-hw-watchpoints

查看断点

列出当前所设置了的所有观察点：
info watchpoints

watch 所设置的断点也可以用控制断点的命令来控制。如 disable、enable、delete 等

贡献者

nanxiao

设置观察点只针对特定线程生效

例子

```
#include <stdio.h>
#include <pthread.h>

int a = 0;

void *thread1_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

void *thread2_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");
        pthread_create(&t2, NULL, thread2_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

gdb可以使用“`watch expr thread threadnum`”命令设置观察点只针对特定线程生效，也就是只有编号为`threadnum`的线程改变了变量的值，程序才会停下来，其它编号线程改变变量的值不会让程序停住。以上面程序为例:

```
(gdb) start
Temporary breakpoint 1 at 0x4005d4: file a.c, line 28.
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Temporary breakpoint 1, main () at a.c:28
28                pthread_create(&t1, NULL, thread1_func, "Thread 1")
(gdb) n
[New Thread 0x7ffff782c700 (LWP 25443)]
29                pthread_create(&t2, NULL, thread2_func, "Thread 2")
(gdb)
[New Thread 0x7ffff6e2b700 (LWP 25444)]
31                sleep(1000);
(gdb) i threads
  Id   Target Id         Frame
  3    Thread 0x7ffff6e2b700 (LWP 25444) 0x00007ffff7915911 in clon
  2    Thread 0x7ffff782c700 (LWP 25443) 0x00007ffff78d9bcd in nanc
* 1    Thread 0x7ffff7fe9700 (LWP 25413) main () at a.c:31
(gdb) wa a thread 2
Hardware watchpoint 2: a
(gdb) c
Continuing.
[Switching to Thread 0x7ffff782c700 (LWP 25443)]
Hardware watchpoint 2: a

Old value = 1
New value = 3
thread1_func (p_arg=0x400718) at a.c:11
11                        sleep(10);
(gdb) c
Continuing.
Hardware watchpoint 2: a

Old value = 3
New value = 5
thread1_func (p_arg=0x400718) at a.c:11
11                        sleep(10);
(gdb) c
Continuing.
Hardware watchpoint 2: a

Old value = 5
New value = 7
thread1_func (p_arg=0x400718) at a.c:11
11                        sleep(10);
```

可以看到，使用“`wa a thread 2`”命令（`wa`是`watch`命令的缩写）以后，只有`thread1_func`改变`a`的值才会让程序停下来。
需要注意的是这种针对特定线程设置观察点方式只对硬件观察点才生效，参见gdb

手册.

贡献者

nanxiao

设置读观察点

例子

```
#include <stdio.h>
#include <pthread.h>

int a = 0;

void *thread1_func(void *p_arg)
{
        while (1)
        {
                printf("%d\n", a);
                sleep(10);
        }
}

int main(void)
{
        pthread_t t1;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");

        sleep(1000);
        return;
}
```

技巧

gdb可以使用“`rwatch`”命令设置读观察点，也就是当发生读取变量行为时，程序就会暂停住。以上面程序为例:

```
(gdb) start
Temporary breakpoint 1 at 0x4005f3: file a.c, line 19.
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Temporary breakpoint 1, main () at a.c:19
19              pthread_create(&t1, NULL, thread1_func, "Thread 1")
(gdb) rw a
Hardware read watchpoint 2: a
(gdb) c
Continuing.
[New Thread 0x7ffff782c700 (LWP 5540)]
[Switching to Thread 0x7ffff782c700 (LWP 5540)]
Hardware read watchpoint 2: a

Value = 0
0x00000000004005c6 in thread1_func (p_arg=0x40071c) at a.c:10
10                      printf("%d\n", a);
(gdb) c
Continuing.
0
Hardware read watchpoint 2: a

Value = 0
0x00000000004005c6 in thread1_func (p_arg=0x40071c) at a.c:10
10                      printf("%d\n", a);
(gdb) c
Continuing.
0
Hardware read watchpoint 2: a

Value = 0
0x00000000004005c6 in thread1_func (p_arg=0x40071c) at a.c:10
10                      printf("%d\n", a);
```

可以看到，使用“`rw a`”命令（`rw`是`rwatch`命令的缩写）以后，每次访问`a`的值都会让程序停下来。
需要注意的是`rwatch`命令只对硬件观察点才生效，参见gdb手册.

贡献者

nanxiao

设置读写观察点

例子

```
#include <stdio.h>
#include <pthread.h>

int a = 0;

void *thread1_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

void *thread2_func(void *p_arg)
{
        while (1)
        {
                printf("%d\n", a);;
                sleep(10);
        }
}

int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");
        pthread_create(&t2, NULL, thread2_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

gdb可以使用“`awatch`”命令设置读写观察点，也就是当发生读取变量或改变变量值的行为时，程序就会暂停住。以上面程序为例:

```
(gdb) aw a
Hardware access (read/write) watchpoint 1: a
(gdb) r
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff782c700 (LWP 16938)]
[Switching to Thread 0x7ffff782c700 (LWP 16938)]
Hardware access (read/write) watchpoint 1: a

Value = 0
0x00000000004005c6 in thread1_func (p_arg=0x40076c) at a.c:10
10                          a++;
(gdb) c
Continuing.
Hardware access (read/write) watchpoint 1: a

Old value = 0
New value = 1
thread1_func (p_arg=0x40076c) at a.c:11
11                          sleep(10);
(gdb) c
Continuing.
[New Thread 0x7ffff6e2b700 (LWP 16939)]
[Switching to Thread 0x7ffff6e2b700 (LWP 16939)]
Hardware access (read/write) watchpoint 1: a

Value = 1
0x00000000004005f2 in thread2_func (p_arg=0x400775) at a.c:19
19                          printf("%d\n", a);;
(gdb) c
Continuing.
1
[Switching to Thread 0x7ffff782c700 (LWP 16938)]
Hardware access (read/write) watchpoint 1: a

Value = 1
0x00000000004005c6 in thread1_func (p_arg=0x40076c) at a.c:10
10                          a++;
```

可以看到，使用“`aw a`”命令（`aw`是`awatch`命令的缩写）以后，每次读取或改变`a`的值都会让程序停下来。
需要注意的是`awatch`命令只对硬件观察点才生效，参见gdb手册.

贡献者

nanxiao

Catchpoint

让catchpoint只触发一次

例子

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    pid_t pid;
    int i = 0;

    for (i = 0; i < 2; i++)
    {
        pid = fork();
        if (pid < 0)
        {
            exit(1);
        }
        else if (pid == 0)
        {
            exit(0);
        }
    }
    printf("hello world\n");
    return 0;
}
```

技巧

使用gdb调试程序时，可以用“`tcatch`”命令设置`catchpoint`只触发一次，以上面程序为例：

```
(gdb) tcatch fork
Catchpoint 1 (fork)
(gdb) r
Starting program: /home/nan/a

Temporary catchpoint 1 (forked process 27377), 0x00000034e42acdbd i
(gdb) c
Continuing.
hello world
[Inferior 1 (process 27373) exited normally]
(gdb) q
```

可以看到当程序只在第一次调用 `fork` 时暂停。

参见gdb手册.

贡献者

nanxiao

为fork调用设置catchpoint

例子

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0)
    {
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0);
    }
    printf("hello world\n");
    return 0;
}
```

技巧

使用gdb调试程序时，可以用“ `catch fork` ”命令为 `fork` 调用设置 `catchpoint` ，以上面程序为例：

```
(gdb) catch fork
Catchpoint 1 (fork)
(gdb) r
Starting program: /home/nan/a

Catchpoint 1 (forked process 33499), 0x00000034e42acdbd in fork ()
(gdb) bt
#0  0x00000034e42acdbd in fork () from /lib64/libc.so.6
#1  0x0000000000400561 in main () at a.c:9
```

可以看到当 `fork` 调用发生后，gdb会暂停程序的运行。
注意：目前只有HP-UX和GNU/Linux支持这个功能。
参见gdb手册.

贡献者

nanxiao

为vfork调用设置catchpoint

例子

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    pid_t pid;

    pid = vfork();
    if (pid < 0)
    {
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0);
    }
    printf("hello world\n");
    return 0;
}
```

技巧

使用gdb调试程序时，可以用“`catch vfork`”命令为`vfork`调用设置`catchpoint`，以上面程序为例：

```
(gdb) catch vfork
Catchpoint 1 (vfork)
(gdb) r
Starting program: /home/nan/a

Catchpoint 1 (vforked process 27312), 0x00000034e42acfc4 in vfork (
   from /lib64/libc.so.6
(gdb) bt
#0  0x00000034e42acfc4 in vfork () from /lib64/libc.so.6
#1  0x0000000000400561 in main () at a.c:9
```

可以看到当 `vfork` 调用发生后，gdb会暂停程序的运行。
注意：目前只有HP-UX和GNU/Linux支持这个功能。
参见gdb手册.

贡献者

nanxiao

为exec调用设置catchpoint

例子

```
#include <unistd.h>

int main(void) {
    execl("/bin/ls", "ls", NULL);
    return 0;
}
```

技巧

使用gdb调试程序时，可以用“`catch exec`”命令为`exec`系列系统调用设置`catchpoint`，以上面程序为例：

```
(gdb) catch exec
Catchpoint 1 (exec)
(gdb) r
Starting program: /home/nan/a
process 32927 is executing new program: /bin/ls

Catchpoint 1 (exec'd /bin/ls), 0x00000034e3a00b00 in _start () fror
(gdb) bt
#0  0x00000034e3a00b00 in _start () from /lib64/ld-linux-x86-64.so
#1  0x0000000000000001 in ?? ()
#2  0x00007fffffffe73d in ?? ()
#3  0x0000000000000000 in ?? ()
```

可以看到当`execl`调用发生后，gdb会暂停程序的运行。
注意：目前只有HP-UX和GNU/Linux支持这个功能。
参见gdb手册.

贡献者

nanxiao

为系统调用设置catchpoint

例子

```
#include <stdio.h>

int main(void)
{
    char p1[] = "Sam";
    char *p2 = "Bob";

    printf("p1 is %s, p2 is %s\n", p1, p2);
    return 0;
}
```

技巧

使用gdb调试程序时，可以使用 `catch syscall [name | number]` 为关注的系统调用设置 `catchpoint` ，以上面程序为例：

```
(gdb) catch syscall mmap
Catchpoint 1 (syscall 'mmap' [9])
(gdb) r
Starting program: /home/nan/a

Catchpoint 1 (call to syscall mmap), 0x00000034e3a16f7a in mmap64 (
   from /lib64/ld-linux-x86-64.so.2
(gdb) c
Continuing.

Catchpoint 1 (returned from syscall mmap), 0x00000034e3a16f7a in mm
   from /lib64/ld-linux-x86-64.so.2
```

可以看到当 `mmap` 调用发生后，gdb会暂停程序的运行。
也可以使用系统调用的编号设置 `catchpoint` ，仍以上面程序为例：

```
(gdb) catch syscall 9
Catchpoint 1 (syscall 'mmap' [9])
(gdb) r
Starting program: /home/nan/a

Catchpoint 1 (call to syscall mmap), 0x00000034e3a16f7a in mmap64 (
   from /lib64/ld-linux-x86-64.so.2
(gdb) c
Continuing.

Catchpoint 1 (returned from syscall mmap), 0x00000034e3a16f7a in mm
   from /lib64/ld-linux-x86-64.so.2
(gdb) c
Continuing.

Catchpoint 1 (call to syscall mmap), 0x00000034e3a16f7a in mmap64 (
   from /lib64/ld-linux-x86-64.so.2
```

可以看到和使用 `catch syscall mmap` 效果是一样的。（系统调用和编号的映射参考具体的 `xml` 文件，以我的系统为例，就是在 `/usr/local/share/gdb/syscalls` 文件夹下的 `amd64-linux.xml` 。）

如果不指定具体的系统调用，则会为所有的系统调用设置 `catchpoint` ，仍以上面程序为例：

```
(gdb) catch syscall
Catchpoint 1 (any syscall)
(gdb) r
Starting program: /home/nan/a

Catchpoint 1 (call to syscall brk), 0x00000034e3a1618a in brk ()
   from /lib64/ld-linux-x86-64.so.2
(gdb) c
Continuing.

Catchpoint 1 (returned from syscall brk), 0x00000034e3a1618a in brk
   from /lib64/ld-linux-x86-64.so.2
(gdb)
Continuing.

Catchpoint 1 (call to syscall mmap), 0x00000034e3a16f7a in mmap64 (
   from /lib64/ld-linux-x86-64.so.2
```

参见gdb手册.

贡献者

nanxiao

通过为ptrace调用设置catchpoint破解anti-debugging的程序

例子

```
#include <sys/ptrace.h>
#include <stdio.h>

int main()
{
        if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0 ) {
                printf("Gdb is debugging me, exit.\n");
                return 1;
        }
        printf("No debugger, continuing\n");
        return 0;
}
```

技巧

有些程序不想被gdb调试，它们就会在程序中调用“ `ptrace` ”函数，一旦返回失败，就证明程序正在被gdb等类似的程序追踪，所以就直接退出。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x400508: file a.c, line 6.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:6
6                       if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0 ) 
(gdb) n
7                               printf("Gdb is debugging me, exit.
(gdb)
Gdb is debugging me, exit.
8                               return 1;
```

破解这类程序的办法就是为 `ptrace` 调用设置 `catchpoint` ，通过修改 `ptrace` 的返回值，达到目的。仍以上面程序为例：

```
(gdb) catch syscall ptrace
Catchpoint 2 (syscall 'ptrace' [101])
(gdb) r
Starting program: /data2/home/nanxiao/a

Catchpoint 2 (call to syscall ptrace), 0x00007ffff7b2be9c in ptrace
(gdb) c
Continuing.

Catchpoint 2 (returned from syscall ptrace), 0x00007ffff7b2be9c in
(gdb) set $rax = 0
(gdb) c
Continuing.
No debugger, continuing
[Inferior 1 (process 11491) exited normally]
```

可以看到，通过修改`rax`寄存器的值，达到修改返回值的目的，从而让gdb可以继续调试程序（打印"`No debugger, continuing`"）。
详细过程，可以参见这篇文章避開 PTRACE_TRACME 反追蹤技巧.

贡献者

nanxiao

打印

打印ASCII和宽字符字符串

例子

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
        char str1[] = "abcd";
        wchar_t str2[] = L"abcd";

        return 0;
}
```

技巧

用gdb调试程序时，可以使用“`x/s`”命令打印ASCII字符串。以上面程序为例：

```
Temporary breakpoint 1, main () at a.c:6
6               char str1[] = "abcd";
(gdb) n
7               wchar_t str2[] = L"abcd";
(gdb)
9               return 0;
(gdb) x/s str1
0x804779f:      "abcd"
```

可以看到打印出了`str1`字符串的值。

打印宽字符字符串时，要根据宽字符的长度决定如何打印。仍以上面程序为例：

```
Temporary breakpoint 1, main () at a.c:6
6               char str1[] = "abcd";
(gdb) n
7               wchar_t str2[] = L"abcd";
(gdb)
9               return 0;
(gdb) p sizeof(wchar_t)
$1 = 4
(gdb) x/ws str2
0x8047788:      U"abcd"
```

由于当前平台宽字符的长度为4个字节，则用“`x/ws`”命令。如果是2个字节，则用“`x/hs`”命令。

参见gdb手册.

贡献者

nanxiao

打印STL容器中的内容

例子

```
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
  vector<int> vec(10); // 10 zero-initialized elements

  for (int i = 0; i < vec.size(); i++)
    vec[i] = i;

  cout << "vec contains:";
  for (int i = 0; i < vec.size(); i++)
    cout << ' ' << vec[i];
  cout << '\n';

  return 0;
}
```

技巧一

在gdb中，如果要打印C++ STL容器的内容，缺省的显示结果可读性很差：

```
(gdb) p vec
$1 = {<std::_Vector_base<int, std::allocator<int> >> = {
    _M_impl = {<std::allocator<int>> = {<__gnu_cxx::new_allocator<i
          _M_end_of_storage = 0x404038}}, <No data fields>}
```

gdb 7.0之后，可以使用gcc提供的python脚本，来改善显示结果：

```
(gdb) p vec
$1 = std::vector of length 10, capacity 10 = {0, 1, 2, 3, 4, 5, 6,
```

某些发行版(Fedora 11+)，不需要额外的设置工作。可在gdb命令行下验证（若没有显示，可按下文的方法进行设置）。

```
(gdb) info pretty-printer
```

方法如下:

1. 获得python脚本，建议使用gcc默认安装的

   ```
   sudo find / -name "*libstdcxx*"
   ```

2. 若本机查找不到python脚本，建议下载gcc对应版本源码包，相对目录如下

   ```
   gcc-4.8.1/libstdc++-v3/python
   ```

3. 也可直接下载最新版本

   ```
   svn co svn://gcc.gnu.org/svn/gcc/trunk/libstdc++-v3/python
   ```

4. 将如下代码添加到.gdbinit文件中（假设python脚本位于 /home/maude/gdb_printers/ 下）

   ```
   python
   import sys
   sys.path.insert(0, '/home/maude/gdb_printers/python')
   from libstdcxx.v6.printers import register_libstdcxx_printers
   register_libstdcxx_printers (None)
   end
   ```

（源自https://sourceware.org/gdb/wiki/STLSupport）

技巧二

`p vec` 的输出无法阅读，但能给我们提示，从而得到无需脚本支持的技巧：

```
(gdb) p *(vec._M_impl._M_start)@vec.size()
$2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

技巧三

将 dbinit_stl_views 下载下来,，执行命令

```
cat dbinit_stl_views-1.03.txt >> ~/.gdbinit
```

即可 一些常用的容器及其对应的命令关系

```
std::vector  pvector stl_variable
std::list  plist stl_variable T
std::map  pmap stl_variable
std::multimap  pmap stl_variable
std::set  pset stl_variable T
std::multiset  pset stl_variable
std::deque  pdequeue stl_variable
std::stack  pstack stl_variable
std::queue  pqueue stl_variable
std::priority_queue  ppqueue stl_variable
std::bitset  pbitset stl_variable
std::string  pstring stl_variable
std::widestring  pwstring stl_variable
```

更多详情，参考配置中的帮助

贡献者

xmj

xanpeng

enjolras

打印大数组中的内容

例子

```
int main()
{
  int array[201];
  int i;

  for (i = 0; i < 201; i++)
    array[i] = i;

  return 0;
}
```

技巧

在gdb中，如果要打印大数组的内容，缺省最多会显示200个元素：

```
(gdb) p array
$1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 6
  95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
  133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145,
  170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
```

可以使用如下命令，设置这个最大限制数：

```
(gdb) set print elements number-of-elements
```

也可以使用如下命令，设置为没有限制：

```
(gdb) set print elements 0
```

或

```
(gdb) set print elements unlimited
(gdb) p array
$2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 6
  95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
  133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145,
  170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
```

详情参见gdb手册

贡献者

xmj

打印数组中任意连续元素值

例子

```
int main(void)
{
  int array[201];
  int i;

  for (i = 0; i < 201; i++)
    array[i] = i;

  return 0;
}
```

技巧

在gdb中，如果要打印数组中任意连续元素的值，可以使用“`p array[index]@num`”命令（`p`是`print`命令的缩写）。其中`index`是数组索引（从0开始计数），`num`是连续多少个元素。以上面代码为例：

```
(gdb) p array
$8 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 4
  63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 7
  94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 1
  120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132,
  145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157,
  170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
  195, 196, 197, 198, 199...}
(gdb) p array[60]@10
$9 = {60, 61, 62, 63, 64, 65, 66, 67, 68, 69}
```

可以看到打印了`array`数组第60~69个元素的值。
如果要打印从数组开头连续元素的值，也可使用这个命令：“`p *array@num`”：

```
(gdb) p *array@10
$2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

详情参见gdb手册

贡献者

nanxiao

打印数组的索引下标

例子

```
#include <stdio.h>

int num[10] = {
  1 << 0,
  1 << 1,
  1 << 2,
  1 << 3,
  1 << 4,
  1 << 5,
  1 << 6,
  1 << 7,
  1 << 8,
  1 << 9
};

int main (void)
{
  int i;

  for (i = 0; i < 10; i++)
    printf ("num[%d] = %d\n", i, num[i]);

  return 0;
}
```

技巧

在gdb中，当打印一个数组时，缺省是不打印索引下标的：

```
(gdb) p num
$1 = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512}
```

如果要打印索引下标，则可以通过如下命令进行设置：

```
(gdb) set print array-indexes on

(gdb) p num
$2 = {[0] = 1, [1] = 2, [2] = 4, [3] = 8, [4] = 16, [5] = 32, [6] =
```

详情参见gdb手册

贡献者

xmj

打印函数局部变量的值

例子

```
#include <stdio.h>

void fun_a(void)
{
    int a = 0;
    printf("%d\n", a);
}

void fun_b(void)
{
    int b = 1;
    fun_a();
    printf("%d\n", b);
}

void fun_c(void)
{
    int c = 2;
    fun_b();
    printf("%d\n", c);
}

void fun_d(void)
{
    int d = 3;
    fun_c();
    printf("%d\n", d);
}

int main(void)
{
    int var = -1;
    fun_d();
    return 0;
}
```

技巧一

如果要打印函数局部变量的值，可以使用“bt full”命令（bt是backtrace的缩写）。首先我们在函数fun_a里打上断点，当程序断住时，显示调用栈信息：

```
(gdb) bt
#0  fun_a () at a.c:6
#1  0x000109b0 in fun_b () at a.c:12
#2  0x000109e4 in fun_c () at a.c:19
#3  0x00010a18 in fun_d () at a.c:26
#4  0x00010a4c in main () at a.c:33
```

接下来，用“bt full”命令显示各个函数的局部变量值：

```
(gdb) bt full
#0  fun_a () at a.c:6
        a = 0
#1  0x000109b0 in fun_b () at a.c:12
        b = 1
#2  0x000109e4 in fun_c () at a.c:19
        c = 2
#3  0x00010a18 in fun_d () at a.c:26
        d = 3
#4  0x00010a4c in main () at a.c:33
        var = -1
```

也可以使用如下“bt full n”，意思是从内向外显示n个栈桢，及其局部变量，例如：

```
(gdb) bt full 2
#0  fun_a () at a.c:6
        a = 0
#1  0x000109b0 in fun_b () at a.c:12
        b = 1
(More stack frames follow...)
```

而“bt full -n”，意思是从外向内显示n个栈桢，及其局部变量，例如：

```
(gdb) bt full -2
#3  0x00010a18 in fun_d () at a.c:26
        d = 3
#4  0x00010a4c in main () at a.c:33
        var = -1
```

详情参见gdb手册

技巧二

如果只是想打印当前函数局部变量的值，可以使用如下命令：

```
(gdb) info locals
a = 0
```

详情参见gdb手册

贡献者

nanxiao

xmj

打印进程内存信息

技巧

用gdb调试程序时，如果想查看进程的内存映射信息，可以使用“i proc mappings”命令（i是info命令缩写），例如:

```
(gdb) i proc mappings
process 27676 flags:
PR_STOPPED Process (LWP) is stopped
PR_ISTOP Stopped on an event of interest
PR_RLC Run-on-last-close is in effect
PR_MSACCT Microstate accounting enabled
PR_PCOMPAT Micro-state accounting inherited on fork
PR_FAULTED : Incurred a traced hardware fault FLTBPT: Breakpoint t

Mapped address spaces:

    Start Addr   End Addr       Size     Offset   Flags
     0x8046000  0x8047fff     0x2000 0xfffff000 -s--rwx
     0x8050000  0x8050fff     0x1000          0 ----r-x
     0x8060000  0x8060fff     0x1000          0 ----rwx
    0xfee40000 0xfef4efff   0x10f000          0 ----r-x
    0xfef50000 0xfef55fff     0x6000          0 ----rwx
    0xfef5f000 0xfef66fff     0x8000   0x10f000 ----rwx
    0xfef67000 0xfef68fff     0x2000          0 ----rwx
    0xfef70000 0xfef70fff     0x1000          0 ----rwx
    0xfef80000 0xfef80fff     0x1000          0 ---sr--
    0xfef90000 0xfef90fff     0x1000          0 ----rw-
    0xfefa0000 0xfefa0fff     0x1000          0 ----rw-
    0xfefb0000 0xfefb0fff     0x1000          0 ----rwx
    0xfefc0000 0xfefeafff    0x2b000          0 ----r-x
    0xfeff0000 0xfeff0fff     0x1000          0 ----rwx
    0xfeffb000 0xfeffcfff     0x2000    0x2b000 ----rwx
    0xfeffd000 0xfeffdfff     0x1000          0 ----rwx
```

首先输出了进程的flags，接着是进程的内存映射信息。
参见gdb手册.

此外，也可以用"i files"（还有一个同样作用的命令：“i target”）命令，它可以更详细地输出进程的内存信息，包括引用的动态链接库等等，例如：

```
(gdb) i files
Symbols from "/data1/nan/a".
Unix /proc child process:
    Using the running image of child Thread 1 (LWP 1) via /proc.
    While running this, GDB does not access memory from...
Local exec file:
    `/data1/nan/a', file type elf32-i386-sol2.
    Entry point: 0x8050950
    0x080500f4 - 0x08050105 is .interp
    0x08050108 - 0x08050114 is .eh_frame_hdr
    0x08050114 - 0x08050218 is .hash
    0x08050218 - 0x08050418 is .dynsym
    0x08050418 - 0x080507e6 is .dynstr
    0x080507e8 - 0x08050818 is .SUNW_version
    0x08050818 - 0x08050858 is .SUNW_versym
    0x08050858 - 0x08050890 is .SUNW_reloc
    0x08050890 - 0x080508c8 is .rel.plt
    0x080508c8 - 0x08050948 is .plt
    ......
    0xfef5fb58 - 0xfef5fc48 is .dynamic in /usr/lib/libc.so.1
    0xfef5fc80 - 0xfef650e2 is .data in /usr/lib/libc.so.1
    0xfef650e2 - 0xfef650e2 is .bssf in /usr/lib/libc.so.1
    0xfef650e8 - 0xfef65be0 is .picdata in /usr/lib/libc.so.1
    0xfef65be0 - 0xfef666a7 is .data1 in /usr/lib/libc.so.1
    0xfef666a8 - 0xfef680dc is .bss in /usr/lib/libc.so.1
```

参见gdb手册

贡献者

nanxiao

打印静态变量的值

例子

```
/* main.c */
extern void print_var_1(void);
extern void print_var_2(void);

int main(void)
{
  print_var_1();
  print_var_2();
  return 0;
}

/* static-1.c */
#include <stdio.h>

static int var = 1;

void print_var_1(void)
{
  printf("var = %d\n", var);
}

/* static-2.c */
#include <stdio.h>

static int var = 2;

void print_var_2(void)
{
  printf("var = %d\n", var);
}
```

技巧

在gdb中，如果直接打印静态变量，则结果并不一定是你想要的：

```
$ gcc -g main.c static-1.c static-2.c
$ gdb -q ./a.out
(gdb) start
(gdb) p var
$1 = 2

$ gcc -g main.c static-2.c static-1.c
$ gdb -q ./a.out
(gdb) start
(gdb) p var
$1 = 1
```

你可以显式地指定文件名（上下文）：

```
(gdb) p 'static-1.c'::var
$1 = 1
(gdb) p 'static-2.c'::var
$2 = 2
```

详情参见gdb手册

贡献者

xmj

打印变量的类型和所在文件

例子

```
#include <stdio.h>

struct child {
  char name[10];
  enum { boy, girl } gender;
};

struct child he = { "Tom", boy };

int main (void)
{
  static struct child she = { "Jerry", girl };
  printf ("Hello %s %s.\n", he.gender == boy ? "boy" : "girl", he.r
  printf ("Hello %s %s.\n", she.gender == boy ? "boy" : "girl", she
  return 0;
}
```

技巧

在gdb中，可以使用如下命令查看变量的类型：

```
(gdb) whatis he
type = struct child
```

如果想查看详细的类型信息：

```
(gdb) ptype he
type = struct child {
    char name[10];
    enum {boy, girl} gender;
}
```

如果想查看定义该变量的文件：

```
(gdb) i variables he
All variables matching regular expression "he":

File variable.c:
struct child he;

Non-debugging symbols:
0x0000000000402030  she
0x00007ffff7dd3380  __check_rhosts_file
```

哦，gdb会显示所有包含（匹配）该表达式的变量。如果只想查看完全匹配给定名字的变量：

```
(gdb) i variables ^he$
All variables matching regular expression "^he$":

File variable.c:
struct child he;
```

注：`info variables`不会显示局部变量，即使是static的也没有太多的信息。

详情参见gdb手册

贡献者

xmj

打印内存的值

例子

```
#include <stdio.h>

int main(void)
{
        int i = 0;
        char a[100];

        for (i = 0; i < sizeof(a); i++)
        {
                a[i] = i;
        }

        return 0;
}
```

技巧

gdb中使用“`x`”命令来打印内存的值，格式为“`x/nfu addr`”。含义为以`f`格式打印从`addr`开始的`n`个长度单元为`u`的内存值。参数具体含义如下：
a）n：输出单元的个数。
b）f：是输出格式。比如`x`是以16进制形式输出，`o`是以8进制形式输出,等等。
c）u：标明一个单元的长度。`b`是一个`byte`，`h`是两个`byte`（halfword），`w`是四个`byte`（word），`g`是八个`byte`（giant word）。

以上面程序为例：
（1）以16进制格式打印数组前`a`16个byte的值：

```
(gdb) x/16xb a
0x7fffffffe4a0: 0x00    0x01    0x02    0x03    0x04    0x05    0x0
0x7fffffffe4a8: 0x08    0x09    0x0a    0x0b    0x0c    0x0d    0x0
```

（2）以无符号10进制格式打印数组`a`前16个byte的值：

```
(gdb) x/16ub a
0x7fffffffe4a0: 0       1       2       3       4       5       6
0x7fffffffe4a8: 8       9       10      11      12      13      14
```

（3） 以2进制格式打印数组前16个 `a` byte的值：

```
(gdb) x/16tb a
0x7fffffffe4a0: 00000000        00000001        00000010        000
0x7fffffffe4a8: 00001000        00001001        00001010        000
```

（4） 以16进制格式打印数组 `a` 前16个word（4个byte）的值：

```
(gdb) x/16xw a
0x7fffffffe4a0: 0x03020100      0x07060504      0x0b0a0908      0x0
0x7fffffffe4b0: 0x13121110      0x17161514      0x1b1a1918      0x1
0x7fffffffe4c0: 0x23222120      0x27262524      0x2b2a2928      0x2
0x7fffffffe4d0: 0x33323130      0x37363534      0x3b3a3938      0x3
```

参见gdb手册.

贡献者

nanxiao

打印源代码行

例子

```
$ gdb -q `which gdb`
(gdb) l
15
16         You should have received a copy of the GNU General Public
17         along with this program.  If not, see <http://www.gnu.org/
18
19      #include "defs.h"
20      #include "main.h"
21      #include <string.h>
22      #include "interps.h"
23
24      int
```

技巧

如上所示，在gdb中可以使用 `list` （简写为l）命令来显示源代码以及行号。 `list` 命令可以指定行号，函数：

```
(gdb) l 24
(gdb) l main
```

还可以指定向前或向后打印：

```
(gdb) l -
(gdb) l +
```

还可以指定范围：

```
(gdb) l 1,10
```

详情参见gdb手册

贡献者

xmj

每行打印一个结构体成员

例子

```
#include <stdio.h>
#include <pthread.h>

typedef struct
{
        int a;
        int b;
        int c;
        int d;
        pthread_mutex_t mutex;
}ex_st;

int main(void) {
        ex_st st = {1, 2, 3, 4, PTHREAD_MUTEX_INITIALIZER};
        printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
        return 0;
}
```

技巧

默认情况下，gdb以一种“紧凑”的方式打印结构体。以上面代码为例：

```
(gdb) n
15              printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
(gdb) p st
$1 = {a = 1, b = 2, c = 3, d = 4, mutex = {__data = {__lock = 0, __
      __spins = 0, __list = {__prev = 0x0, __next = 0x0}}, __size =
```

可以看到结构体的显示很混乱，尤其是结构体里还嵌套着其它结构体时。

可以执行“set print pretty on”命令，这样每行只会显示结构体的一名成员，而且还会根据成员的定义层次进行缩进：

```
(gdb) set print pretty on
(gdb) p st
$2 = {
  a = 1,
  b = 2,
  c = 3,
  d = 4,
  mutex = {
    __data = {
      __lock = 0,
      __count = 0,
      __owner = 0,
      __nusers = 0,
      __kind = 0,
      __spins = 0,
      __list = {
        __prev = 0x0,
        __next = 0x0
      }
    },
    __size = '\000' <repeats 39 times>,
    __align = 0
  }
}
```

详情参见gdb手册

贡献者

nanxiao

按照派生类型打印对象

例子

```
#include <iostream>
using namespace std;

class Shape {
 public:
  virtual void draw () {}
};

class Circle : public Shape {
 int radius;
 public:
  Circle () { radius = 1; }
  void draw () { cout << "drawing a circle...\n"; }
};

class Square : public Shape {
 int height;
 public:
  Square () { height = 2; }
  void draw () { cout << "drawing a square...\n"; }
};

void drawShape (class Shape &p)
{
  p.draw ();
}

int main (void)
{
  Circle a;
  Square b;
  drawShape (a);
  drawShape (b);
  return 0;
}
```

技巧

在gdb中，当打印一个对象时，缺省是按照声明的类型进行打印：

```
(gdb) frame
#0  drawShape (p=...) at object.cxx:25
25      p.draw ();
(gdb) p p
$1 = (Shape &) @0x7fffffffde90: {_vptr.Shape = 0x400a80 <vtable fo
```

在这个例子中，p虽然声明为class Shape，但它实际的派生类型可能为class Circle和Square。如果要缺省按照派生类型进行打印，则可以通过如下命令进行设置：

```
(gdb) set print object on

(gdb) p p
$2 = (Circle &) @0x7fffffffde90: {<Shape> = {_vptr.Shape = 0x400a8
```

当打印对象类型信息时，该设置也会起作用：

```
(gdb) whatis p
type = Shape &
(gdb) ptype p
type = class Shape {
  public:
    virtual void draw(void);
} &

(gdb) set print object on
(gdb) whatis p
type = /* real type = Circle & */
Shape &
(gdb) ptype p
type = /* real type = Circle & */
class Shape {
  public:
    virtual void draw(void);
} &
```

详情参见gdb手册

贡献者

xmj

xanpeng

指定程序的输入输出设备

例子

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 0; i < 100; i++)
    {
      printf("i = %d\n", i);
    }

  return 0;
}
```

技巧

在gdb中，缺省情况下程序的输入输出是和gdb使用同一个终端。你也可以为程序指定一个单独的输入输出终端。

首先，打开一个新终端，使用如下命令获得设备文件名：

```
$ tty
/dev/pts/2
```

然后，通过命令行选项指定程序的输入输出设备：

```
$ gdb -tty /dev/pts/2 ./a.out
(gdb) r
```

或者，在gdb中，使用命令进行设置：

```
(gdb) tty /dev/pts/2
```

详情参见gdb手册

贡献者

xmj

使用“$_”和“$__”变量

例子

```
#include <stdio.h>

int main(void)
{
        int i = 0;
        char a[100];

        for (i = 0; i < sizeof(a); i++)
        {
                a[i] = i;
        }

        return 0;
}
```

技巧

"`x`"命令会把最后检查的内存地址值存在“`$_`”这个“convenience variable”中，并且会把这个地址中的内容放在“`$__`”这个“convenience variable”，以上面程序为例:

```
(gdb) b a.c:13
Breakpoint 1 at 0x4004a0: file a.c, line 13.
(gdb) r
Starting program: /data2/home/nanxiao/a

Breakpoint 1, main () at a.c:13
13              return 0;
(gdb) x/16xb a
0x7fffffffe4a0: 0x00    0x01    0x02    0x03    0x04    0x05    0x0
0x7fffffffe4a8: 0x08    0x09    0x0a    0x0b    0x0c    0x0d    0x0
(gdb) p $_
$1 = (int8_t *) 0x7fffffffe4af
(gdb) p $__
$2 = 15
```

可以看到“`$_`”值为`0x7fffffffe4af`，正好是"`x`"命令检查的最后的内存地址。而“`$__`”值为`15`。
另外要注意有些命令（像“`info line`”和“`info breakpoint`”）会提供一个默认的地址给"`x`"命令检查，而这些命令也会把“`$_`”的值变为那个默认地址值：

```
(gdb) p $_
$5 = (int8_t *) 0x7fffffffe4af
(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004a0 in main at a.c:1
        breakpoint already hit 1 time
(gdb) p $_
$6 = (void *) 0x4004a0 <main+44>
```

可以看到使用“`info breakpoint`”命令后，“`$_`”值变为`0x4004a0`。
参见gdb手册.

贡献者

nanxiao

打印程序动态分配内存的信息

例子

```
#include <stdio.h>
#include <malloc.h>

int main(void)
{
        char *p[10];
        int i = 0;

        for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
        {
                p[i] = malloc(100000);
        }
        return 0;
}
```

技巧

用gdb调试程序时，可以用下面的自定义命令，打印程序动态分配内存的信息：

```
define mallocinfo
  set $__f = fopen("/dev/tty", "w")
  call malloc_info(0, $__f)
  call fclose($__f)
end
```

以上面程序为例：

```
Temporary breakpoint 5, main () at a.c:7
7               int i = 0;
(gdb) mallocinfo
<malloc version="1">
<heap nr="0">
<sizes>
</sizes>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="135168"/>
<system type="max" size="135168"/>
<aspace type="total" size="135168"/>
```

```
<aspace type="mprotect" size="135168"/>
</heap>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="135168"/>
<system type="max" size="135168"/>
<aspace type="total" size="135168"/>
<aspace type="mprotect" size="135168"/>
</malloc>
$20 = 0
$21 = 0
(gdb) n
9               for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                      p[i] = malloc(100000);
(gdb)
9               for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                      p[i] = malloc(100000);
(gdb)
9               for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                      p[i] = malloc(100000);
(gdb)
9               for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                      p[i] = malloc(100000);
(gdb)
9               for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                      p[i] = malloc(100000);
(gdb) mallocinfo
<malloc version="1">
<heap nr="0">
<sizes>
</sizes>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="532480"/>
<system type="max" size="532480"/>
<aspace type="total" size="532480"/>
<aspace type="mprotect" size="532480"/>
</heap>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="532480"/>
<system type="max" size="532480"/>
<aspace type="total" size="532480"/>
<aspace type="mprotect" size="532480"/>
</malloc>
$22 = 0
$23 = 0
(gdb) n
```

```
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                          p[i] = malloc(100000);
(gdb)
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                          p[i] = malloc(100000);
(gdb)
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                          p[i] = malloc(100000);
(gdb)
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                          p[i] = malloc(100000);
(gdb)
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb)
11                          p[i] = malloc(100000);
(gdb)
9                   for (i = 0; i < sizeof(p)/sizeof(p[0]); i++)
(gdb) mallocinfo
<malloc version="1">
<heap nr="0">
<sizes>
</sizes>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="1134592"/>
<system type="max" size="1134592"/>
<aspace type="total" size="1134592"/>
<aspace type="mprotect" size="1134592"/>
</heap>
<total type="fast" count="0" size="0"/>
<total type="rest" count="0" size="0"/>
<system type="current" size="1134592"/>
<system type="max" size="1134592"/>
<aspace type="total" size="1134592"/>
<aspace type="mprotect" size="1134592"/>
</malloc>
$24 = 0
$25 = 0
```

可以看到gdb输出了动态分配内存的变化信息。
参见stackoverflow.

贡献者

nanxiao

打印调用栈帧中变量的值

例子

```
#include <stdio.h>

int func1(int a)
{
  int b = 1;
  return b * a;
}

int func2(int a)
{
  int b = 2;
  return b * func1(a);
}

int func3(int a)
{
  int b = 3;
  return b * func2(a);
}

int main(void)
{
  printf("%d\n", func3(10));
  return 0;
}
```

技巧

在gdb中，如果想查看调用栈帧中的变量，可以先切换到该栈帧中，然后打印：

```
(gdb) b func1
(gdb) r
(gdb) bt
#0  func1 (a=10) at frame.c:5
#1  0x0000000000400560 in func2 (a=10) at frame.c:12
#2  0x0000000000400582 in func3 (a=10) at frame.c:18
#3  0x0000000000400596 in main () at frame.c:23
(gdb) f 1
(gdb) p b
(gdb) f 2
(gdb) p b
```

也可以不进行切换，直接打印：

```
(gdb) p func2::b
$1 = 2
(gdb) p func3::b
$2 = 3
```

同样，对于C++的函数名，需要使用单引号括起来，比如：

```
(gdb) p '(anonymous namespace)::SSAA::handleStore'::n->pi->inst->du
```

详情参见gdb手册

贡献者

xmj

多进程/线程

调试已经运行的进程

例子

```
#include <stdio.h>
#include <pthread.h>
void *thread_func(void *p_arg)
{
        while (1)
        {
                printf("%s\n", (char*)p_arg);
                sleep(10);
        }
}
int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread_func, "Thread 1");
        pthread_create(&t2, NULL, thread_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

调试已经运行的进程有两种方法：一种是gdb启动时，指定进程的ID：gdb program processID（也可以用-p或者--pid指定进程ID，例如：gdb program -p=10210）。以上面代码为例，用“ps”命令已经获得进程ID为10210：

```
bash-3.2# gdb -q a 10210
Reading symbols from /data/nan/a...done.
Attaching to program `/data/nan/a', process 10210
[New process 10210]
Retry #1:
Retry #2:
Retry #3:
Retry #4:
Reading symbols from /usr/lib/libc.so.1...(no debugging symbols fou
[Thread debugging using libthread_db enabled]
[New LWP    3        ]
[New LWP    2        ]
[New Thread 1 (LWP 1)]
[New Thread 2 (LWP 2)]
[New Thread 3 (LWP 3)]
Loaded symbols for /usr/lib/libc.so.1
Reading symbols from /lib/ld.so.1...(no debugging symbols found)..
Loaded symbols for /lib/ld.so.1
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /usr/lib/libc.so.1
(gdb) bt
#0  0xfeeeae55 in ___nanosleep () from /usr/lib/libc.so.1
#1  0xfeedcae4 in sleep () from /usr/lib/libc.so.1
#2  0x080509ef in main () at a.c:17
```

如果嫌每次ps查看进程号比较麻烦，请尝试如下脚本

```
# 保存为xgdb.sh（添加可执行权限）
# 用法 xgdb.sh a
prog_bin=$1
running_name=$(basename $prog_bin)
pid=$(/sbin/pidof $running_name)
gdb attach $pid
```

另一种是先启动gdb，然后用“attach”命令“附着”在进程上：

```
bash-3.2# gdb -q a
Reading symbols from /data/nan/a...done.
(gdb) attach 10210
Attaching to program `/data/nan/a', process 10210
[New process 10210]
Retry #1:
Retry #2:
Retry #3:
Retry #4:
Reading symbols from /usr/lib/libc.so.1...(no debugging symbols fou
[Thread debugging using libthread_db enabled]
[New LWP    3        ]
[New LWP    2        ]
[New Thread 1 (LWP 1)]
[New Thread 2 (LWP 2)]
[New Thread 3 (LWP 3)]
Loaded symbols for /usr/lib/libc.so.1
Reading symbols from /lib/ld.so.1...(no debugging symbols found)..
Loaded symbols for /lib/ld.so.1
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /usr/lib/libc.so.1
(gdb) bt
#0  0xfeeeae55 in ___nanosleep () from /usr/lib/libc.so.1
#1  0xfeedcae4 in sleep () from /usr/lib/libc.so.1
#2  0x080509ef in main () at a.c:17
```

如果不想继续调试了，可以用“detach”命令“脱离”进程：

```
(gdb) detach
Detaching from program: /data/nan/a, process 10210
(gdb) bt
No stack.
```

详情参见gdb手册

贡献者

nanxiao

调试子进程

例子

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0)
    {
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0);
    }
    printf("hello world\n");
    return 0;
}
```

技巧

在调试多进程程序时，gdb默认会追踪父进程。例如：

```
(gdb) start
Temporary breakpoint 1 at 0x40055c: file a.c, line 8.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:8
8               pid = fork();
(gdb) n
9               if (pid < 0)
(gdb) hello world

13              else if (pid > 0)
(gdb)
15                      exit(0);
(gdb)
[Inferior 1 (process 12786) exited normally]
```

可以看到程序执行到第15行：父进程退出。

如果要调试子进程，要使用如下命令："set follow-fork-mode child"，例如：

```
(gdb) set follow-fork-mode child
(gdb) start
Temporary breakpoint 1 at 0x40055c: file a.c, line 8.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:8
8               pid = fork();
(gdb) n
[New process 12241]
[Switching to process 12241]
9               if (pid < 0)
(gdb)
13              else if (pid > 0)
(gdb)
17              printf("hello world\n");
(gdb)
hello world
18              return 0;
```

可以看到程序执行到第17行：子进程打印"hello world"。

这个命令目前Linux支持，其它很多操作系统都不支持，使用时请注意。参见gdb手册

贡献者

nanxiao

同时调试父进程和子进程

例子

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0)
    {
        exit(1);
    }
    else if (pid > 0)
    {
        printf("Parent\n");
        exit(0);
    }
    printf("Child\n");
    return 0;
}
```

技巧

在调试多进程程序时，gdb默认只会追踪父进程的运行，而子进程会独立运行，gdb不会控制。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x40055c: file a.c, line 7.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:7
7           pid = fork();
(gdb) n
8           if (pid < 0)
(gdb) Child

12          else if (pid > 0)
(gdb)
14              printf("Parent\n");
(gdb)
Parent
15              exit(0);
```

可以看到当单步执行到第8行时，程序打印出“Child” ，证明子进程已经开始独立运行。

如果要同时调试父进程和子进程，可以使用“ `set detach-on-fork off` ”（默认 `detach-on-fork` 是 `on` ）命令，这样gdb就能同时调试父子进程，并且在调试一个进程时，另外一个进程处于挂起状态。仍以上面程序为例：

```
(gdb) set detach-on-fork off
(gdb) start
Temporary breakpoint 1 at 0x40055c: file a.c, line 7.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:7
7           pid = fork();
(gdb) n
[New process 1050]
8           if (pid < 0)
(gdb)
12          else if (pid > 0)
(gdb) i inferior
  Num  Description       Executable
  2    process 1050      /data2/home/nanxiao/a
* 1    process 1046      /data2/home/nanxiao/a
(gdb) n
14              printf("Parent\n");
(gdb) n
Parent
15              exit(0);
(gdb)
[Inferior 1 (process 1046) exited normally]
(gdb)
The program is not being run.
(gdb) i inferiors
  Num  Description       Executable
  2    process 1050      /data2/home/nanxiao/a
* 1    <null>            /data2/home/nanxiao/a
(gdb) inferior 2
[Switching to inferior 2 [process 1050] (/data2/home/nanxiao/a)]
[Switching to thread 2 (process 1050)]
#0  0x00007ffff7af6cad in fork () from /lib64/libc.so.6
(gdb) bt
#0  0x00007ffff7af6cad in fork () from /lib64/libc.so.6
#1  0x0000000000400561 in main () at a.c:7
(gdb) n
Single stepping until exit from function fork,
which has no line number information.
main () at a.c:8
8           if (pid < 0)
(gdb)
12          else if (pid > 0)
(gdb)
17          printf("Child\n");
(gdb)
Child
18          return 0;
(gdb)
```

在使用“ `set detach-on-fork off` ”命令后，用“ `i inferiors` ”（ `i` 是 `info` 命令缩写）查看进程状态，可以看到父子进程都在被gdb调试的状态，前面显示“*”是正在调试的进程。当父进程退出后，用“ `inferior infno` ”切换到子进程去调试。

这个命令目前Linux支持，其它很多操作系统都不支持，使用时请注意。参见gdb手册

此外，如果想让父子进程都同时运行，可以使用“ `set schedule-multiple on` ”（默认 `schedule-multiple` 是 `off` ）命令，仍以上述代码为例：

```
(gdb) set detach-on-fork off
(gdb) set schedule-multiple on
(gdb) start
Temporary breakpoint 1 at 0x40059c: file a.c, line 7.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:7
7               pid = fork();
(gdb) n
[New process 26597]
Child
```

可以看到打印出了“Child”，证明子进程也在运行了。
参见gdb手册

贡献者

nanxiao

查看线程信息

例子

```
#include <stdio.h>
#include <pthread.h>
void *thread_func(void *p_arg)
{
        while (1)
        {
                printf("%s\n", (char*)p_arg);
                sleep(10);
        }
}
int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread_func, "Thread 1");
        pthread_create(&t2, NULL, thread_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

用gdb调试多线程程序，可以用“i threads”命令（i是info命令缩写）查看所有线程的信息，以上面程序为例（运行平台为Linux，CPU为X86_64）:

```
  (gdb) i threads
  Id   Target Id         Frame
  3    Thread 0x7ffff6e2b700 (LWP 31773) 0x00007ffff7915911 in clor
  2    Thread 0x7ffff782c700 (LWP 31744) 0x00007ffff78d9bcd in nanc
* 1    Thread 0x7ffff7fe9700 (LWP 31738) main () at a.c:18
```

第一项（Id）：是gdb标示每个线程的唯一ID：1，2等等。
第二项（Target Id）：是具体系统平台用来标示每个线程的ID，不同平台信息可能会不同。 像当前Linux平台显示的就是： Thread 0x7ffff6e2b700 (LWP 31773)。
第三项（Frame）：显示的是线程执行到哪个函数。
前面带“*”表示的是“current thread”，可以理解为gdb调试多线程程序时，选择的一个“默认线程”。

再以Solaris平台（CPU为X86_64）为例，可以看到显示信息会略有不同：

```
(gdb) i threads
[New Thread 2 (LWP 2)]
[New Thread 3 (LWP 3)]
  Id   Target Id         Frame
  6    Thread 3 (LWP 3)  0xfeec870d in _thr_setup () from /usr/lib/
  5    Thread 2 (LWP 2)  0xfefc9661 in elf_find_sym () from /usr/li
  4    LWP    3          0xfeec870d in _thr_setup () from /usr/lib/
  3    LWP    2          0xfefc9661 in elf_find_sym () from /usr/li
* 2    Thread 1 (LWP 1)  main () at a.c:18
  1    LWP    1          main () at a.c:18
```

也可以用“i threads [Id...]”指定打印某些线程的信息，例如：

```
  (gdb) i threads 1 2
  Id   Target Id         Frame
  2    Thread 0x7ffff782c700 (LWP 12248) 0x00007ffff78d9bcd in nanc
* 1    Thread 0x7ffff7fe9700 (LWP 12244) main () at a.c:18
```

参见gdb手册.

贡献者

nanxiao

在Solaris上使用maintenance命令查看线程信息

技巧

用gdb调试多线程程序时，如果想查看线程信息，可以使用“i threads”命令（i是info命令缩写），例如:

```
(gdb) i threads
106 process 2689429      0xff04af84 in __lwp_park () from /lib/libc
105 process 2623893      0xff04af84 in __lwp_park () from /lib/libc
104 process 2558357      0xff04af84 in __lwp_park () from /lib/libc
103 process 2492821      0xff04af84 in __lwp_park () from /lib/libc
```

在Solaris操作系统上，gdb为Solaris量身定做了一个查看线程信息的命令：“maint info sol-threads”（maint是maintenance命令缩写），例如:

```
(gdb) maint info sol-threads
user    thread #1, lwp 1, (active)
user    thread #2, lwp 2, (active)    startfunc: monitor_thread
user    thread #3, lwp 3, (asleep)    startfunc: mem_db_thread
- Sleep func: 0x000aa32c
```

可以看到相比于info命令，maintenance命令显示了更多信息。例如线程当前状态（active，asleep），入口函数（startfunc）等。

参见gdb手册

贡献者

nanxiao

不显示线程启动和退出信息

例子

```
#include <stdio.h>
#include <pthread.h>

void *thread_func(void *p_arg)
{
       sleep(10);
}

int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread_func, "Thread 1");
        pthread_create(&t2, NULL, thread_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

默认情况下，gdb检测到有线程产生和退出时，会打印提示信息，以上面程序为例:

```
(gdb) r
Starting program: /data/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[New LWP    2        ]
[New LWP    3        ]
[LWP    2         exited]
[New Thread 2        ]
[LWP    3         exited]
[New Thread 3        ]
```

如果不想显示这些信息，可以使用“`set print thread-events off`”命令，这样当有线程产生和退出时，就不会打印提示信息：

```
(gdb) set print thread-events off
(gdb) r
Starting program: /data/nan/a
[Thread debugging using libthread_db enabled]
```

可以看到不再打印相关信息。

这个命令有些平台不支持，使用时需注意。参见gdb手册.

贡献者

nanxiao

只允许一个线程运行

例子

```
#include <stdio.h>
#include <pthread.h>
int a = 0;
int b = 0;
void *thread1_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(1);
        }
}

void *thread2_func(void *p_arg)
{
        while (1)
        {
                b++;
                sleep(1);
        }
}

int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");
        pthread_create(&t2, NULL, thread2_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

用gdb调试多线程程序时，一旦程序断住，所有的线程都处于暂停状态。此时当你调试其中一个线程时（比如执行“`step`”，“`next`”命令），所有的线程都会同时执行。以上面程序为例:

```
(gdb) b a.c:9
Breakpoint 1 at 0x400580: file a.c, line 9.
(gdb) r
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff782c700 (LWP 17368)]
[Switching to Thread 0x7ffff782c700 (LWP 17368)]

Breakpoint 1, thread1_func (p_arg=0x400718) at a.c:9
9                       a++;
(gdb) p b
$1 = 0
(gdb) s
10                      sleep(1);
(gdb) s
[New Thread 0x7ffff6e2b700 (LWP 17369)]
11              }
(gdb)

Breakpoint 1, thread1_func (p_arg=0x400718) at a.c:9
9                       a++;
(gdb)
10                      sleep(1);
(gdb) p b
$2 = 3
```

`thread1_func`更新全局变量`a`的值，`thread2_func`更新全局变量`b`的值。我在`thread1_func`里`a++`语句打上断点，当断点第一次命中时，打印`b`的值是`0`，在单步调试`thread1_func`几次后，`b`的值变成`3`，证明在单步调试`thread1_func`时，`thread2_func`也在执行。
如果想在调试一个线程时，让其它线程暂停执行，可以使用“`set scheduler-locking on`”命令：

```
(gdb) b a.c:9
Breakpoint 1 at 0x400580: file a.c, line 9.
(gdb) r
Starting program: /data2/home/nanxiao/a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff782c700 (LWP 19783)]
[Switching to Thread 0x7ffff782c700 (LWP 19783)]

Breakpoint 1, thread1_func (p_arg=0x400718) at a.c:9
9                       a++;
(gdb) set scheduler-locking on
(gdb) p b
$1 = 0
(gdb) s
10                      sleep(1);
(gdb)
11              }
(gdb)

Breakpoint 1, thread1_func (p_arg=0x400718) at a.c:9
9                       a++;
(gdb)
10                      sleep(1);
(gdb)
11              }
(gdb) p b
$2 = 0
```

可以看到在单步调试`thread1_func`几次后，`b`的值仍然为`0`，证明在在单步调试`thread1_func`时，`thread2_func`没有执行。

此外，“`set scheduler-locking`”命令除了支持`off`和`on`模式外（默认是`off`），还有一个`step`模式。含义是：当用"`step`"命令调试线程时，其它线程不会执行，但是用其它命令（比如"`next`"）调试线程时，其它线程也许会执行。

这个命令依赖于具体操作系统的调度策略，使用时需注意。参见gdb手册.

贡献者

nanxiao

使用“$_thread”变量

例子

```
#include <stdio.h>
#include <pthread.h>

int a = 0;

void *thread1_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

void *thread2_func(void *p_arg)
{
        while (1)
        {
                a++;
                sleep(10);
        }
}

int main(void)
{
        pthread_t t1, t2;

        pthread_create(&t1, NULL, thread1_func, "Thread 1");
        pthread_create(&t2, NULL, thread2_func, "Thread 2");

        sleep(1000);
        return;
}
```

技巧

gdb从7.2版本引入了`$_thread`这个“`convenience variable`”，用来保存当前正在调试的线程号。这个变量在写断点命令或是命令脚本时会很有用。以上面程序为例:

```
(gdb) wa a
Hardware watchpoint 2: a
(gdb) command 2
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>printf "thread id=%d\n", $_thread
>end
```

首先设置了观察点："wa a"（`wa`是`watch`命令缩写），也就是当`a`的值发生变化时，程序会暂停，接下来在`commands`语句中打印线程号。
然后继续执行程序：

```
(gdb) c
Continuing.
[New Thread 0x7ffff782c700 (LWP 20928)]
[Switching to Thread 0x7ffff782c700 (LWP 20928)]
Hardware watchpoint 2: a

Old value = 0
New value = 1
thread1_func (p_arg=0x400718) at a.c:11
11                      sleep(10);
thread id=2
(gdb) c
Continuing.
[New Thread 0x7ffff6e2b700 (LWP 20929)]
[Switching to Thread 0x7ffff6e2b700 (LWP 20929)]
Hardware watchpoint 2: a

Old value = 1
New value = 2
thread2_func (p_arg=0x400721) at a.c:20
20                      sleep(10);
thread id=3
```

可以看到程序暂停时，会打印线程号："`thread id=2`"或者"`thread id=3`"。
参见gdb手册.

贡献者

nanxiao

一个gdb会话中同时调试多个程序

例子

```
a.c:
#include <stdio.h>
int func(int a, int b)
{
        int c = a * b;
        printf("c is %d\n", c);
}

int main(void)
{
        func(1, 2);
        return 0;
}


b.c:
#include <stdio.h>

int func1(int a)
{
        return 2 * a;
}

int func2(int a)
{
        int c = 0;
        c = 2 * func1(a);
        return c;
}

int func3(int a)
{
        int c = 0;
        c = 2 * func2(a);
        return c;
}

int main(void)
{
        printf("%d\n", func3(10));
        return 0;
}
```

技巧

gdb支持在一个会话中同时调试多个程序。以上面程序为例，首先调试`a`程序：

```
root@bash:~$ gdb a
GNU gdb (Ubuntu 7.7-0ubuntu3) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show c
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x400568: file a.c, line 10.
Starting program: /home/nanxiao/a
```

接着使用“`add-inferior [ -copies n ] [ -exec executable ]`”命令加载可执行文件`b`。其中`n`默认为1：

```
(gdb) add-inferior -copies 2 -exec b
Added inferior 2
Reading symbols from b...done.
Added inferior 3
Reading symbols from b...done.
(gdb) i inferiors
  Num  Description       Executable
  3    <null>            /home/nanxiao/b
  2    <null>            /home/nanxiao/b
* 1    process 1586      /home/nanxiao/a
(gdb) inferior 2
[Switching to inferior 2 [<null>] (/home/nanxiao/b)]
(gdb) start
Temporary breakpoint 2 at 0x400568: main. (3 locations)
Starting program: /home/nanxiao/b

Temporary breakpoint 2, main () at b.c:24
24                printf("%d\n", func3(10));
(gdb) i inferiors
  Num  Description       Executable
  3    <null>            /home/nanxiao/b
* 2    process 1590      /home/nanxiao/b
  1    process 1586      /home/nanxiao/a
```

可以看到可以调试`b`程序了。

另外也可用"`clone-inferior [ -copies n ] [ infno ]`"克隆现有的`inferior`，其中`n`默认为1，`infno`默认为当前的`inferior`：

```
(gdb) i inferiors
  Num  Description       Executable
  3    <null>            /home/nanxiao/b
* 2    process 1590      /home/nanxiao/b
  1    process 1586      /home/nanxiao/a
(gdb) clone-inferior -copies 1
Added inferior 4.
(gdb) i inferiors
  Num  Description       Executable
  4    <null>            /home/nanxiao/b
  3    <null>            /home/nanxiao/b
* 2    process 1590      /home/nanxiao/b
  1    process 1586      /home/nanxiao/a
```

可以看到又多了一个`b`程序。

参见gdb手册.

贡献者

nanxiao

打印程序进程空间信息

例子

```
a.c:
#include <stdio.h>
int func(int a, int b)
{
        int c = a * b;
        printf("c is %d\n", c);
}

int main(void)
{
        func(1, 2);
        return 0;
}


b.c:
#include <stdio.h>

int func1(int a)
{
        return 2 * a;
}

int func2(int a)
{
        int c = 0;
        c = 2 * func1(a);
        return c;
}

int func3(int a)
{
        int c = 0;
        c = 2 * func2(a);
        return c;
}

int main(void)
{
        printf("%d\n", func3(10));
        return 0;
}
```

技巧

使用gdb调试多个进程时，可以使用“`maint info program-spaces`”打印当前所有被调试的进程信息。以上面程序为例：

```
[root@localhost nan]# gdb a
GNU gdb (GDB) 7.8.1
......
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x4004f9: file a.c, line 10.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:10
10              func(1, 2);
(gdb) add-inferior -exec b
Added inferior 2
Reading symbols from b...done.
(gdb) i inferiors b
Args must be numbers or '$' variables.
(gdb) i inferiors
  Num  Description       Executable
  2    <null>            /home/nan/b
* 1    process 15753     /home/nan/a
(gdb) inferior 2
[Switching to inferior 2 [<null>] (/home/nan/b)]
(gdb) start
Temporary breakpoint 2 at 0x4004f9: main. (2 locations)
Starting program: /home/nan/b

Temporary breakpoint 2, main () at b.c:24
24              printf("%d\n", func3(10));
(gdb) i inferiors
  Num  Description       Executable
* 2    process 15902     /home/nan/b
  1    process 15753     /home/nan/a
(gdb) clone-inferior -copies 2
Added inferior 3.
Added inferior 4.
(gdb) i inferiors
  Num  Description       Executable
  4    <null>            /home/nan/b
  3    <null>            /home/nan/b
* 2    process 15902     /home/nan/b
  1    process 15753     /home/nan/a
(gdb) maint info program-spaces
  Id   Executable
  4    /home/nan/b
        Bound inferiors: ID 4 (process 0)
  3    /home/nan/b
        Bound inferiors: ID 3 (process 0)
* 2    /home/nan/b
        Bound inferiors: ID 2 (process 15902)
  1    /home/nan/a
        Bound inferiors: ID 1 (process 15753)
```

可以看到执行“`maint info program-spaces`”命令后，打印出当前有4个`program-spaces`（编号从1到4）。另外还有每个`program-spaces`对应的程序，`inferior`编号及进程号。

参见gdb手册.

贡献者

nanxiao

使用“$_exitcode”变量

例子

```
int main(void)
{
    return 0;
}
```

技巧

当被调试的程序正常退出时，gdb会使用`$_exitcode`这个“`convenience variable`”记录程序退出时的“`exit code`”。以调试上面程序为例:

```
[root@localhost nan]# gdb -q a
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x400478: file a.c, line 3.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:3
3                return 0;
(gdb) n
4       }
(gdb)
0x00000034e421ed1d in __libc_start_main () from /lib64/libc.so.6
(gdb)
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 1185) exited normally]
(gdb) p $_exitcode
$1 = 0
```

可以看到打印的`$_exitcode`的值为`0`。
改变程序，返回值改为`1`：

```
int main(void)
{
    return 0;
}
```

接着调试：

```
[root@localhost nan]# gdb -q a
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x400478: file a.c, line 3.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:3
3               return 1;
(gdb)
(gdb) n
4       }
(gdb)
0x00000034e421ed1d in __libc_start_main () from /lib64/libc.so.6
(gdb)
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 2603) exited with code 01]
(gdb) p $_exitcode
$1 = 1
```

可以看到打印的`$_exitcode`的值变为`1`。
参见gdb手册.

贡献者

nanxiao

core dump文件

为调试进程产生core dump文件

技巧

在用gdb调试程序时，我们有时想让被调试的进程产生core dump文件，记录现在进程的状态，以供以后分析。可以用“generate-core-file”命令来产生core dump文件：

```
(gdb) help generate-core-file
Save a core file with the current state of the debugged process.
Argument is optional filename.  Default filename is 'core.<process_

(gdb) start
Temporary breakpoint 1 at 0x8050c12: file a.c, line 9.
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 1, main () at a.c:9
9           change_var();
(gdb) generate-core-file
Saved corefile core.12955
```

也可使用“gcore”命令：

```
(gdb) help gcore
Save a core file with the current state of the debugged process.
Argument is optional filename.  Default filename is 'core.<process_
(gdb) gcore
Saved corefile core.13256
```

参见gdb手册

贡献者

nanxiao

加载可执行程序和core dump文件

例子

```
#include <stdio.h>

int main(void) {
        int *p = NULL;
        printf("hello world\n");
        *p = 0;
        return 0;
}
```

技巧

例子程序访问了一个空指针，所以程序会crash并产生core dump文件。用gdb调试core dump文件，通常用这个命令形式："gdb path/to/the/executable path/to/the/coredump"，然后gdb会显示程序crash的位置：

```
bash-3.2# gdb -q /data/nan/a /var/core/core.a.22268.1402638140
Reading symbols from /data/nan/a...done.
[New LWP 1]
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
Core was generated by `./a'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400cdb in main () at a.c:6
6               *p = 0;
```

有时我们想在gdb启动后，动态加载可执行程序和core dump文件，这时可以用"file"和"core"（core-file命令缩写）命令。"file"命令用来读取可执行文件的符号表信息，而"core"命令则是指定core dump文件的位置：

```
bash-3.2# gdb -q
(gdb) file /data/nan/a
Reading symbols from /data/nan/a...done.
(gdb) core /var/core/core.a.22268.1402638140
[New LWP 1]
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
Core was generated by `./a'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400cdb in main () at a.c:6
6               *p = 0;
```

可以看到gdb同样显示程序crash的位置。

这两个命令可参见gdb手册

贡献者

nanxiao

汇编

设置汇编指令格式

例子

```
#include <stdio.h>
int global_var;

void change_var(){
    global_var=100;
}

int main(void){
    change_var();
    return 0;
}
```

技巧

在Intel x86处理器上，gdb默认显示汇编指令格式是AT&T格式。例如：

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08050c0f <+0>:     push   %ebp
   0x08050c10 <+1>:     mov    %esp,%ebp
   0x08050c12 <+3>:     call   0x8050c00 <change_var>
   0x08050c17 <+8>:     mov    $0x0,%eax
   0x08050c1c <+13>:    pop    %ebp
   0x08050c1d <+14>:    ret
End of assembler dump.
```

可以用“set disassembly-flavor”命令将格式改为intel格式：

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x08050c0f <+0>:     push   ebp
   0x08050c10 <+1>:     mov    ebp,esp
   0x08050c12 <+3>:     call   0x8050c00 <change_var>
   0x08050c17 <+8>:     mov    eax,0x0
   0x08050c1c <+13>:    pop    ebp
   0x08050c1d <+14>:    ret
End of assembler dump.
```

目前“set disassembly-flavor”命令只能用在Intel x86处理器上，并且取值只有“intel”和“att”。

详情参见gdb手册

贡献者

nanxiao

在函数的第一条汇编指令打断点

例子

```
#include <stdio.h>
int global_var;

void change_var(){
    global_var=100;
}

int main(void){
    change_var();
    return 0;
}
```

技巧

通常给函数打断点的命令："b func"（b是break命令的缩写），不会把断点设置在汇编指令层次函数的开头，例如：

```
(gdb) b main
Breakpoint 1 at 0x8050c12: file a.c, line 9.
(gdb) r
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Breakpoint 1, main () at a.c:9
9           change_var();
(gdb) disassemble
Dump of assembler code for function main:
   0x08050c0f <+0>:     push   %ebp
   0x08050c10 <+1>:     mov    %esp,%ebp
=> 0x08050c12 <+3>:     call   0x8050c00 <change_var>
   0x08050c17 <+8>:     mov    $0x0,%eax
   0x08050c1c <+13>:    pop    %ebp
   0x08050c1d <+14>:    ret
End of assembler dump.
```

可以看到程序停在了第三条汇编指令（箭头所指位置）。如果要把断点设置在汇编指令层次函数的开头，要使用如下命令："b *func"，例如：

```
(gdb) b *main
Breakpoint 1 at 0x8050c0f: file a.c, line 8.
(gdb) r
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Breakpoint 1, main () at a.c:8
8       int main(void){
(gdb) disassemble
Dump of assembler code for function main:
=> 0x08050c0f <+0>:     push   %ebp
   0x08050c10 <+1>:     mov    %esp,%ebp
   0x08050c12 <+3>:     call   0x8050c00 <change_var>
   0x08050c17 <+8>:     mov    $0x0,%eax
   0x08050c1c <+13>:    pop    %ebp
   0x08050c1d <+14>:    ret
End of assembler dump.
```

可以看到程序停在了第一条汇编指令（箭头所指位置）。

贡献者

nanxiao

自动反汇编后面要执行的代码

例子

```
(gdb) set disassemble-next-line on
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Temporary breakpoint 3 at 0x400543: file 1.c, line 14.
Starting program: /home/teawater/tmp/a.out

Temporary breakpoint 3, main (argc=1, argv=0x7fffffffdf38, envp=0x7
14      printf("1\n");
=> 0x0000000000400543 <main+19>:    bf f0 05 40 00  mov    $0x4005f
   0x0000000000400548 <main+24>:    e8 c3 fe ff ff  callq  0x40041(
(gdb) si
0x0000000000400548  14      printf("1\n");
0x0000000000400543 <main+19>:    bf f0 05 40 00  mov    $0x4005f0,%
=> 0x0000000000400548 <main+24>:    e8 c3 fe ff ff  callq  0x40041(
(gdb)
0x0000000000400410 in puts@plt ()
=> 0x0000000000400410 <puts@plt+0>: ff 25 02 0c 20 00   jmpq   *0x2

(gdb) set disassemble-next-line auto
(gdb) start
Temporary breakpoint 1 at 0x400543: file 1.c, line 14.
Starting program: /home/teawater/tmp/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdf38, envp=0x7
14      printf("1\n");
(gdb) si
0x0000000000400548  14      printf("1\n");
(gdb)
0x0000000000400410 in puts@plt ()
=> 0x0000000000400410 <puts@plt+0>: ff 25 02 0c 20 00   jmpq   *0x2
(gdb)
0x0000000000400416 in puts@plt ()
=> 0x0000000000400416 <puts@plt+6>: 68 00 00 00 00  pushq  $0x0
```

技巧

如果要在任意情况下反汇编后面要执行的代码：

```
(gdb) set disassemble-next-line on
```

如果要在后面的代码没有源码的情况下才反汇编后面要执行的代码：

```
(gdb) set disassemble-next-line auto
```

关闭这个功能：

```
(gdb) set disassemble-next-line off
```

贡献者

teawater

将源程序和汇编指令映射起来

例子

```
#include <stdio.h>

typedef struct
{
        int a;
        int b;
        int c;
        int d;
}ex_st;

int main(void) {
        ex_st st = {1, 2, 3, 4};
        printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
        return 0;
}
```

技巧一

可以用“disas /m fun”（disas是disassemble命令缩写）命令将函数代码和汇编指令映射起来，以上面代码为例：

```
(gdb) disas /m main
Dump of assembler code for function main:
11      int main(void) {
   0x00000000004004c4 <+0>:     push   %rbp
   0x00000000004004c5 <+1>:     mov    %rsp,%rbp
   0x00000000004004c8 <+4>:     push   %rbx
   0x00000000004004c9 <+5>:     sub    $0x18,%rsp

12              ex_st st = {1, 2, 3, 4};
   0x00000000004004cd <+9>:     movl   $0x1,-0x20(%rbp)
   0x00000000004004d4 <+16>:    movl   $0x2,-0x1c(%rbp)
   0x00000000004004db <+23>:    movl   $0x3,-0x18(%rbp)
   0x00000000004004e2 <+30>:    movl   $0x4,-0x14(%rbp)

13              printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
   0x00000000004004e9 <+37>:    mov    -0x14(%rbp),%esi
   0x00000000004004ec <+40>:    mov    -0x18(%rbp),%ecx
   0x00000000004004ef <+43>:    mov    -0x1c(%rbp),%edx
   0x00000000004004f2 <+46>:    mov    -0x20(%rbp),%ebx
   0x00000000004004f5 <+49>:    mov    $0x400618,%eax
   0x00000000004004fa <+54>:    mov    %esi,%r8d
   0x00000000004004fd <+57>:    mov    %ebx,%esi
   0x00000000004004ff <+59>:    mov    %rax,%rdi
   0x0000000000400502 <+62>:    mov    $0x0,%eax
   0x0000000000400507 <+67>:    callq  0x4003b8 <printf@plt>

14              return 0;
   0x000000000040050c <+72>:    mov    $0x0,%eax

15      }
   0x0000000000400511 <+77>:    add    $0x18,%rsp
   0x0000000000400515 <+81>:    pop    %rbx
   0x0000000000400516 <+82>:    leaveq
   0x0000000000400517 <+83>:    retq

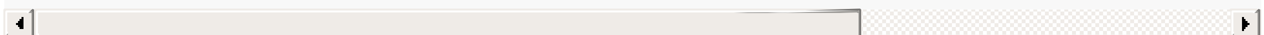
End of assembler dump.
```

可以看到每一条C语句下面是对应的汇编代码。

技巧二

如果只想查看某一行所对应的地址范围，可以：

```
(gdb) i line 13
Line 13 of "foo.c" starts at address 0x4004e9 <main+37> and ends at
```



如果只想查看这一条语句对应的汇编代码，可以使用“ `disassemble [Start],[End]` ”命令：

```
(gdb) disassemble 0x4004e9, 0x40050c
Dump of assembler code from 0x4004e9 to 0x40050c:
   0x00000000004004e9 <main+37>:        mov    -0x14(%rbp),%esi
   0x00000000004004ec <main+40>:        mov    -0x18(%rbp),%ecx
   0x00000000004004ef <main+43>:        mov    -0x1c(%rbp),%edx
   0x00000000004004f2 <main+46>:        mov    -0x20(%rbp),%ebx
   0x00000000004004f5 <main+49>:        mov    $0x400618,%eax
   0x00000000004004fa <main+54>:        mov    %esi,%r8d
   0x00000000004004fd <main+57>:        mov    %ebx,%esi
   0x00000000004004ff <main+59>:        mov    %rax,%rdi
   0x0000000000400502 <main+62>:        mov    $0x0,%eax
   0x0000000000400507 <main+67>:        callq  0x4003b8 <printf@plt
End of assembler dump.
```

详情参见gdb手册

贡献者

nanxiao

xmj

显示将要执行的汇编指令

例子

```
#include <stdio.h>
int global_var;

void change_var(){
    global_var=100;
}

int main(void){
    change_var();
    return 0;
}
```

技巧

使用gdb调试汇编程序时，可以用“`display /i $pc`”命令显示当程序停止时，将要执行的汇编指令。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x400488: file a.c, line 9.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:9
9            change_var();
(gdb) display /i $pc
1: x/i $pc
=> 0x400488 <main+4>:   mov    $0x0,%eax
(gdb) si
0x000000000040048d      9           change_var();
1: x/i $pc
=> 0x40048d <main+9>:   callq  0x400474 <change_var>
(gdb)
change_var () at a.c:4
4        void change_var(){
1: x/i $pc
=> 0x400474 <change_var>:       push   %rbp
```

可以看到打印出了将要执行的汇编指令。此外也可以一次显示多条指令：

```
(gdb) display /3i $pc
2: x/3i $pc
=> 0x400474 <change_var>:       push   %rbp
   0x400475 <change_var+1>:     mov    %rsp,%rbp
   0x400478 <change_var+4>:     movl   $0x64,0x2003de(%rip)
```

可以看到一次显示了`3`条指令。

取消显示可以用`undisplay`命令。

详情参见gdb手册

贡献者

nanxiao

打印寄存器的值

技巧

用gdb调试程序时，如果想查看寄存器的值，可以使用“i registers”命令（i是info命令缩写），例如:

```
(gdb) i registers
rax            0x7ffff7dd9f60   140737351884640
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe608   140737488348680
rsi            0x7fffffffe5f8   140737488348664
rdi            0x1      1
rbp            0x7fffffffe510   0x7fffffffe510
rsp            0x7fffffffe4c0   0x7fffffffe4c0
r8             0x7ffff7dd8300   140737351877376
r9             0x7ffff7deb9e0   140737351956960
r10            0x7fffffffe360   140737488348000
r11            0x7ffff7a68be0   140737348275168
r12            0x4003e0 4195296
r13            0x7fffffffe5f0   140737488348656
r14            0x0      0
r15            0x0      0
rip            0x4004cd 0x4004cd <main+9>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

以上输出不包括浮点寄存器和向量寄存器的内容。使用“i all-registers”命令，可以输出所有寄存器的内容：

```
(gdb) i all-registers
    rax            0x7ffff7dd9f60   140737351884640
    rbx            0x0      0
    rcx            0x0      0
    rdx            0x7fffffffe608   140737488348680
    rsi            0x7fffffffe5f8   140737488348664
    rdi            0x1      1
    rbp            0x7fffffffe510   0x7fffffffe510
    rsp            0x7fffffffe4c0   0x7fffffffe4c0
    r8             0x7ffff7dd8300   140737351877376
    r9             0x7ffff7deb9e0   140737351956960
    r10            0x7fffffffe360   140737488348000
    r11            0x7ffff7a68be0   140737348275168
    r12            0x4003e0 4195296
    r13            0x7fffffffe5f0   140737488348656
    r14            0x0      0
    r15            0x0      0
    rip            0x4004cd 0x4004cd <main+9>
    eflags         0x206    [ PF IF ]
    cs             0x33     51
    ss             0x2b     43
    ds             0x0      0
    es             0x0      0
    fs             0x0      0
    gs             0x0      0
    st0            0        (raw 0x00000000000000000000)
    st1            0        (raw 0x00000000000000000000)
    st2            0        (raw 0x00000000000000000000)
    st3            0        (raw 0x00000000000000000000)
    st4            0        (raw 0x00000000000000000000)
    st5            0        (raw 0x00000000000000000000)
    st6            0        (raw 0x00000000000000000000)
    st7            0        (raw 0x00000000000000000000)
    ......
```

要打印单个寄存器的值，可以使用“i registers regname”或者“p $regname”，例如：

```
(gdb) i registers eax
eax            0xf7dd9f60       -136470688
(gdb) p $eax
$1 = -136470688
```

参见gdb手册.

贡献者

nanxiao

显示程序原始机器码

例子

```
#include <stdio.h>

int main(void)
{
        printf("Hello, world\n");
        return 0;
}
```

技巧

使用“disassemble /r”命令可以用16进制形式显示程序的原始机器码。以上面程序为例：

```
(gdb) disassemble /r main
Dump of assembler code for function main:
   0x0000000000400530 <+0>:     55      push   %rbp
   0x0000000000400531 <+1>:     48 89 e5        mov    %rsp,%rbp
   0x0000000000400534 <+4>:     bf e0 05 40 00  mov    $0x4005e0,%e
   0x0000000000400539 <+9>:     e8 d2 fe ff ff  callq  0x400410 <pu
   0x000000000040053e <+14>:    b8 00 00 00 00  mov    $0x0,%eax
   0x0000000000400543 <+19>:    5d      pop    %rbp
   0x0000000000400544 <+20>:    c3      retq
End of assembler dump.
(gdb) disassemble /r 0x0000000000400534,+4
Dump of assembler code from 0x400534 to 0x400538:
   0x0000000000400534 <main+4>: bf e0 05 40 00  mov    $0x4005e0,%e
End of assembler dump.
```

详情参见gdb手册

贡献者

nanxiao

改变程序的执行

改变字符串的值

例子

```
#include <stdio.h>

int main(void)
{
    char p1[] = "Sam";
    char *p2 = "Bob";

    printf("p1 is %s, p2 is %s\n", p1, p2);
    return 0;
}
```

技巧

使用gdb调试程序时，可以用“`set`”命令改变字符串的值，以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x8050af0: file a.c, line 5.
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 1, main () at a.c:5
5               char p1[] = "Sam";
(gdb) n
6               char *p2 = "Bob";
(gdb)
8               printf("p1 is %s, p2 is %s\n", p1, p2);
(gdb) set main::p1="Jil"
(gdb) set main::p2="Bill"
(gdb) n
p1 is Jil, p2 is Bill
9               return 0;
```

可以看到执行`p1`和`p2`的字符串都发生了变化。也可以通过访问内存地址的方法改变字符串的值：

```
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 2, main () at a.c:5
5               char p1[] = "Sam";
(gdb) n
6               char *p2 = "Bob";
(gdb) p p1
$1 = "Sam"
(gdb) p &p1
$2 = (char (*)[4]) 0x80477a4
(gdb) set {char [4]} 0x80477a4 = "Ace"
(gdb) n
8               printf("p1 is %s, p2 is %s\n", p1, p2);
(gdb)
p1 is Ace, p2 is Bob
9               return 0;
```

在改变字符串的值时候，一定要注意内存越界的问题。
参见stackoverflow.

贡献者

nanxiao

设置变量的值

例子

```
#include <stdio.h>

int func(void)
{
    int i = 2;

    return i;
}

int main(void)
{
    int a = 0;

    a = func();
    printf("%d\n", a);
    return 0;
}
```

技巧

在gdb中，可以用“`set var variable=expr`”命令设置变量的值，以上面代码为例：

```
Breakpoint 2, func () at a.c:5
5                   int i = 2;
(gdb) n
7                   return i;
(gdb) set var i = 8
(gdb) p i
$4 = 8
```

可以看到在`func`函数里用`set`命令把`i`的值修改成为`8`。

也可以用“`set {type}address=expr`”的方式，含义是给存储地址在`address`，变量类型为`type`的变量赋值，仍以上面代码为例：

```
Breakpoint 2, func () at a.c:5
5                   int i = 2;
(gdb) n
7                   return i;
(gdb) p &i
$5 = (int *) 0x8047a54
(gdb) set {int}0x8047a54 = 8
(gdb) p i
$6 = 8
```

可以看到 `i` 的值被修改成为 `8` 。

另外寄存器也可以作为变量，因此同样可以修改寄存器的值：

```
Breakpoint 2, func () at a.c:5
5                   int i = 2;
(gdb)
(gdb) n
7                   return i;
(gdb)
8               }
(gdb) set var $eax = 8
(gdb) n
main () at a.c:15
15                  printf("%d\n", a);
(gdb)
8
16                  return 0;
```

可以看到因为eax寄存器存储着函数的返回值，所以当把eax寄存器的值改为 `8` 后，函数的返回值也变成了 `8` 。

详情参见gdb手册

贡献者

nanxiao

修改PC寄存器的值

例子

```
#include <stdio.h>
int main(void)
{
        int a =0;

        a++;
        a++;
        printf("%d\n", a);
        return 0;
}
```

技巧

PC寄存器会存储程序下一条要执行的指令，通过修改这个寄存器的值，可以达到改变程序执行流程的目的。
上面的程序会输出“`a=2`”，下面介绍一下如何通过修改PC寄存器的值，改变程序执行流程。

```
4               int a =0;
(gdb) disassemble main
Dump of assembler code for function main:
0x08050921 <main+0>:    push   %ebp
0x08050922 <main+1>:    mov    %esp,%ebp
0x08050924 <main+3>:    sub    $0x8,%esp
0x08050927 <main+6>:    and    $0xfffffff0,%esp
0x0805092a <main+9>:    mov    $0x0,%eax
0x0805092f <main+14>:   add    $0xf,%eax
0x08050932 <main+17>:   add    $0xf,%eax
0x08050935 <main+20>:   shr    $0x4,%eax
0x08050938 <main+23>:   shl    $0x4,%eax
0x0805093b <main+26>:   sub    %eax,%esp
0x0805093d <main+28>:   movl   $0x0,-0x4(%ebp)
0x08050944 <main+35>:   lea    -0x4(%ebp),%eax
0x08050947 <main+38>:   incl   (%eax)
0x08050949 <main+40>:   lea    -0x4(%ebp),%eax
0x0805094c <main+43>:   incl   (%eax)
0x0805094e <main+45>:   sub    $0x8,%esp
0x08050951 <main+48>:   pushl  -0x4(%ebp)
0x08050954 <main+51>:   push   $0x80509b4
0x08050959 <main+56>:   call   0x80507cc <printf@plt>
0x0805095e <main+61>:   add    $0x10,%esp
0x08050961 <main+64>:   mov    $0x0,%eax
0x08050966 <main+69>:   leave
0x08050967 <main+70>:   ret
End of assembler dump.
(gdb) info line 6
Line 6 of "a.c" starts at address 0x8050944 <main+35> and ends at (
(gdb) info line 7
Line 7 of "a.c" starts at address 0x8050949 <main+40> and ends at (
```

通过“ `info line 6` ”和“ `info line 7` ”命令可以知道两条“ `a++;` ”语句的汇编指令起始地址分别是 `0x8050944` 和 `0x8050949` 。

```
(gdb) n
6               a++;
(gdb) p $pc
$3 = (void (*)()) 0x8050944 <main+35>
(gdb) set var $pc=0x08050949
```

当程序要执行第一条“ `a++;` ”语句时，打印 `pc` 寄存器的值，看到 `pc` 寄存器的值为 `0x8050944` ，与“ `info line 6` ”命令得到的一致。接下来，把 `pc` 寄存器的值改为 `0x8050949` ，也就是通过“ `info line 7` ”命令得到的第二条“ `a++;` ”语句的起始地址。

```
(gdb) n
8               printf("a=%d\n", a);
(gdb)
a=1
9               return 0;
```

接下来执行，可以看到程序输出“`a=1`”，也就是跳过了第一条“`a++;`”语句。

贡献者

nanxiao

跳转到指定位置执行

例子

```
#include <stdio.h>

void fun (int x)
{
  if (x < 0)
    puts ("error");
}

int main (void)
{
  int i = 1;

  fun (i--);
  fun (i--);
  fun (i--);

  return 0;
}
```

技巧

当调试程序时，你可能不小心走过了出错的地方：

```
(gdb) n
13        fun (i--);
(gdb)
14        fun (i--);
(gdb)
15        fun (i--);
(gdb)
error
17        return 0;
```

看起来是在15行，调用fun的时候出错了。常见的办法是在15行设置个断点，然后从头 `run` 一次。

如果你的环境支持反向执行，那么更好了。

如果不支持，你也可以直接 `jump` 到15行，再执行一次：

```
(gdb) b 15
Breakpoint 2 at 0x40056a: file jump.c, line 15.
(gdb) j 15
Continuing at 0x40056a.

Breakpoint 2, main () at jump.c:15
15        fun (i--);
(gdb) s
fun (x=-2) at jump.c:5
5       if (x < 0)
(gdb) n
6         puts ("error");
```

需要注意的是：

1. `jump` 命令只改变pc的值，所以改变程序执行可能会出现不同的结果，比如变量i的值
2. 通过（临时）断点的配合，可以让你的程序跳到指定的位置，并停下来

详情参见gdb手册

贡献者

xmj

使用断点命令改变程序的执行

例子

```
#include <stdio.h>
#include <stdlib.h>

void drawing (int n)
{
  if (n != 0)
    puts ("Try again?\nAll you need is a dollar, and a dream.");
  else
    puts ("You win $3000!");
}

int main (void)
{
  int n;

  srand (time (0));
  n = rand () % 10;
  printf ("Your number is %d\n", n);
  drawing (n);

  return 0;
}
```

技巧

这个例子程序可能不太好，只是可以用来演示下断点命令的用法：

```
(gdb) b drawing
Breakpoint 1 at 0x40064d: file win.c, line 6.
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>silent
>set variable n = 0
>continue
>end
(gdb) r
Starting program: /home/xmj/tmp/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_dk
Your number is 6
You win $3000!
[Inferior 1 (process 4134) exited normally]
```

可以看到，当程序运行到断点处，会自动把变量n的值修改为0，然后继续执行。

如果你在调试一个大程序，重新编译一次会花费很长时间，比如调试编译器的bug，那么你可以用这种方式在gdb中先实验性的修改下试试，而不需要修改源码，重新编译。

详情参见gdb手册

贡献者

xmj

修改被调试程序的二进制文件

例子

```
#include <stdio.h>
#include <stdlib.h>

void drawing (int n)
{
  if (n != 0)
    puts ("Try again?\nAll you need is a dollar, and a dream.");
  else
    puts ("You win $3000!");
}

int main (void)
{
  int n;

  srand (time (0));
  n = rand () % 10;
  printf ("Your number is %d\n", n);
  drawing (n);

  return 0;
}
```

技巧

gdb不仅可以用来调试程序，还可以修改程序的二进制代码。

缺省情况下，gdb是以只读方式加载程序的。可以通过命令行选项指定为可写：

```
$ gcc -write ./a.out
(gdb) show write
Writing into executable and core files is on.
```

也可以在gdb中，使用命令设置并重新加载程序：

```
(gdb) set write on
(gdb) file ./a.out
```

接下来，查看反汇编：

```
(gdb) disassemble /mr drawing
Dump of assembler code for function drawing:
5     {
   0x0000000000400642 <+0>:    55    push   %rbp
   0x0000000000400643 <+1>:    48 89 e5     mov    %rsp,%rbp
   0x0000000000400646 <+4>:    48 83 ec 10     sub    $0x10,%rsp
   0x000000000040064a <+8>:    89 7d fc     mov    %edi,-0x4(%rbp)

6       if (n != 0)
   0x000000000040064d <+11>:    83 7d fc 00     cmpl   $0x0,-0x4(%rb
   0x0000000000400651 <+15>:    74 0c    je     0x40065f <drawing+2

7         puts ("Try again?\nAll you need is a dollar, and a dream.'
   0x0000000000400653 <+17>:    bf e0 07 40 00     mov    $0x4007e0,
   0x0000000000400658 <+22>:    e8 b3 fe ff ff     callq  0x400510 <
   0x000000000040065d <+27>:    eb 0a    jmp    0x400669 <drawing+3

8       else
9         puts ("You win $3000!");
   0x000000000040065f <+29>:    bf 12 08 40 00     mov    $0x400812,
   0x0000000000400664 <+34>:    e8 a7 fe ff ff     callq  0x400510 <

10    }
   0x0000000000400669 <+39>:    c9    leaveq
   0x000000000040066a <+40>:    c3    retq

End of assembler dump.
```

修改二进制代码（注意大小端和指令长度）：

```
(gdb) set variable *(short*)0x400651=0x0ceb
(gdb) disassemble /mr drawing
Dump of assembler code for function drawing:
5    {
   0x0000000000400642 <+0>:     55      push   %rbp
   0x0000000000400643 <+1>:     48 89 e5        mov    %rsp,%rbp
   0x0000000000400646 <+4>:     48 83 ec 10     sub    $0x10,%rsp
   0x000000000040064a <+8>:     89 7d fc        mov    %edi,-0x4(%rbp)

6       if (n != 0)
   0x000000000040064d <+11>:    83 7d fc 00     cmpl   $0x0,-0x4(%rb
   0x0000000000400651 <+15>:    eb 0c   jmp    0x40065f <drawing+2

7         puts ("Try again?\nAll you need is a dollar, and a dream.'
   0x0000000000400653 <+17>:    bf e0 07 40 00  mov    $0x4007e0,
   0x0000000000400658 <+22>:    e8 b3 fe ff ff  callq  0x400510 <
   0x000000000040065d <+27>:    eb 0a   jmp    0x400669 <drawing+3

8       else
9         puts ("You win $3000!");
   0x000000000040065f <+29>:    bf 12 08 40 00  mov    $0x400812,
   0x0000000000400664 <+34>:    e8 a7 fe ff ff  callq  0x400510 <

10    }
   0x0000000000400669 <+39>:    c9      leaveq
   0x000000000040066a <+40>:    c3      retq

End of assembler dump.
```

可以看到，条件跳转指令“je”已经被改为无条件跳转“jmp”了。

退出，运行一下：

```
$ ./a.out
Your number is 2
You win $3000!
```

详情参见gdb手册

贡献者

xmj

信号

查看信号处理信息

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGINT, handler);
        signal(SIGALRM, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

用gdb调试程序时，可以用“`i signals`”命令（或者“`i handle`”命令，`i`是`info`命令缩写）查看gdb如何处理进程收到的信号:

```
(gdb) i signals
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
SIGINT        Yes       Yes     No              Interrupt
SIGQUIT       Yes       Yes     Yes             Quit
......
SIGALRM       No        No      Yes             Alarm clock
......
```

第一项（ `Signal` ）：标示每个信号。
第二项（ `Stop` ）：表示被调试的程序有对应的信号发生时，gdb是否会暂停程序。
第三项（ `Print` ）：表示被调试的程序有对应的信号发生时，gdb是否会打印相关信息。
第四项（ `Pass to program` ）：gdb是否会把这个信号发给被调试的程序。
第五项（ `Description` ）：信号的描述信息。

从上面的输出可以看到，当 `SIGINT` 信号发生时，gdb会暂停被调试的程序，并打印相关信息，但不会把这个信号发给被调试的程序。而当 `SIGALRM` 信号发生时，gdb不会暂停被调试的程序，也不打印相关信息，但会把这个信号发给被调试的程序。

启动gdb调试上面的程序，同时另起一个终端，先后发送 `SIGINT` 和 `SIGALRM` 信号给被调试的进程，输出如下：

```
Program received signal SIGINT, Interrupt.
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) c
Continuing.
Receive signal: 14
```

可以看到收到 `SIGINT` 时，程序暂停了，也输出了信号信息，但并没有把 `SIGINT` 信号交由进程处理（程序没有输出）。而收到 `SIGALRM` 信号时，程序没有暂停，也没有输出信号信息，但把 `SIGALRM` 信号交由进程处理了（程序打印了输出）。

参见gdb手册.

贡献者

nanxiao

信号发生时是否暂停程序

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGHUP, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

用gdb调试程序时，可以用“`handle signal stop/nostop`”命令设置当信号发生时，是否暂停程序的执行，以上面程序为例:

```
(gdb) i signals
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
......

(gdb) r
Starting program: /data1/nan/test
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]

Program received signal SIGHUP, Hangup.
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) c
Continuing.
Receive signal: 1
```

可以看到，默认情况下，发生`SIGHUP`信号时，gdb会暂停程序的执行，并打印收到信号的信息。此时需要执行`continue`命令继续程序的执行。

接下来用“`handle SIGHUP nostop`”命令设置当`SIGHUP`信号发生时，gdb不暂停程序，执行如下：

```
(gdb) handle SIGHUP nostop
Signal        Stop      Print   Pass to program Description
SIGHUP        No        Yes     Yes             Hangup
(gdb) c
Continuing.

Program received signal SIGHUP, Hangup.
Receive signal: 1
```

可以看到，程序收到`SIGHUP`信号发生时，没有暂停，而是继续执行。

如果想恢复之前的行为，用“`handle SIGHUP stop`”命令即可。需要注意的是，设置`stop`的同时，默认也会设置`print`（关于`print`，请参见信号发生时是否打印信号信息）。

参见gdb手册.

贡献者

nanxiao

信号发生时是否打印信号信息

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGHUP, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

用gdb调试程序时，可以用“`handle signal print/noprint`”命令设置当信号发生时，是否打印信号信息，以上面程序为例:

```
(gdb) i signals
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
......

(gdb) r
Starting program: /data1/nan/test
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]

Program received signal SIGHUP, Hangup.
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) c
Continuing.
Receive signal: 1
```

可以看到，默认情况下，发生 `SIGHUP` 信号时，gdb会暂停程序的执行，并打印收到信号的信息。此时需要执行 `continue` 命令继续程序的执行。

接下来用“ `handle SIGHUP noprint` ”命令设置当 `SIGHUP` 信号发生时，gdb不打印信号信息，执行如下：

```
(gdb) handle SIGHUP noprint
Signal        Stop      Print   Pass to program Description
SIGHUP        No        No      Yes             Hangup
(gdb) r
Starting program: /data1/nan/test
[Thread debugging using libthread_db enabled]
Receive signal: 1
```

需要注意的是，设置 `noprint` 的同时，默认也会设置 `nostop` 。可以看到，程序收到 `SIGHUP` 信号发生时，没有暂停，也没有打印信号信息。而是继续执行。

如果想恢复之前的行为，用“ `handle SIGHUP print` ”命令即可。

参见gdb手册.

贡献者

nanxiao

信号发生时是否把信号丢给程序处理

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGHUP, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

用gdb调试程序时，可以用“`handle signal pass(noignore)/nopass(ignore)`”命令设置当信号发生时，是否把信号丢给程序处理.其中`pass`和`noignore`含义相同，`nopass`和`ignore`含义相同。以上面程序为例:

```
(gdb) i signals
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
......

(gdb) r
Starting program: /data1/nan/test
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]

Program received signal SIGHUP, Hangup.
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) c
Continuing.
Receive signal: 1
```

可以看到，默认情况下，发生 `SIGHUP` 信号时，gdb会把信号丢给程序处理。

接下来用“ `handle SIGHUP nopass` ”命令设置当 `SIGHUP` 信号发生时，gdb不把信号丢给程序处理，执行如下：

```
(gdb) handle SIGHUP nopass
Signal        Stop      Print   Pass to program Description
SIGHUP        Yes       Yes     No              Hangup
(gdb) c
Continuing.

Program received signal SIGHUP, Hangup.
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) c
Continuing.
```

可以看到， `SIGHUP` 信号发生时，程序没有打印“Receive signal: 1”，说明gdb没有把信号丢给程序处理。

如果想恢复之前的行为，用“ `handle SIGHUP pass` ”命令即可。

参见gdb手册.

贡献者

nanxiao

给程序发送信号

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGHUP, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

用gdb调试程序的过程中，当被调试程序停止后，可以用“`signal signal_name`”命令让程序继续运行，但会立即给程序发送信号。以上面程序为例:

```
(gdb) r
`/data1/nan/test' has changed; re-reading symbols.
Starting program: /data1/nan/test
[Thread debugging using libthread_db enabled]
^C[New Thread 1 (LWP 1)]

Program received signal SIGINT, Interrupt.
[Switching to Thread 1 (LWP 1)]
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) signal SIGHUP
Continuing with signal SIGHUP.
Receive signal: 1
```

可以看到，当程序暂停后，执行`signal SIGHUP`命令，gdb会发送信号给程序处理。

可以使用“`signal 0`”命令使程序重新运行，但不发送任何信号给进程。仍以上面程序为例：

```
Program received signal SIGHUP, Hangup.
0xfeeeae55 in ___nanosleep () from /lib/libc.so.1
(gdb) signal 0
Continuing with no signal.
```

可以看到，`SIGHUP`信号发生时，gdb停住了程序，但是由于执行了“`signal 0`”命令，所以程序重新运行后，并没有收到`SIGHUP`信号。

使用`signal`命令和在shell环境使用`kill`命令给程序发送信号的区别在于：在shell环境使用`kill`命令给程序发送信号，gdb会根据当前的设置决定是否把信号发送给进程，而使用`signal`命令则直接把信号发给进程。

参见gdb手册.

贡献者

nanxiao

使用“$_siginfo”变量

例子

```
#include <stdio.h>
#include <signal.h>

void handler(int sig);

void handler(int sig)
{
        signal(sig, handler);
        printf("Receive signal: %d\n", sig);
}

int main(void) {
        signal(SIGHUP, handler);

        while (1)
        {
                sleep(1);
        }
        return 0;
}
```

技巧

在某些平台上（比如Linux）使用gdb调试程序，当有信号发生时，gdb在把信号丢给程序之前，可以通过 `$_siginfo` 变量读取一些额外的有关当前信号的信息，这些信息是 `kernel` 传给信号处理函数的。以上面程序为例:

```
Program received signal SIGHUP, Hangup.
0x00000034e42accc0 in __nanosleep_nocancel () from /lib64/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.13
(gdb) ptype $_siginfo
type = struct {
    int si_signo;
    int si_errno;
    int si_code;
    union {
        int _pad[28];
        struct {...} _kill;
        struct {...} _timer;
        struct {...} _rt;
        struct {...} _sigchld;
        struct {...} _sigfault;
        struct {...} _sigpoll;
    } _sifields;
}
(gdb) ptype $_siginfo._sifields._sigfault
type = struct {
    void *si_addr;
}
(gdb) p $_siginfo._sifields._sigfault.si_addr
$1 = (void *) 0x850e
```

我们可以了解 `$_siginfo` 变量里每个成员的类型，并且可以读到成员的值。

参见gdb手册.

贡献者

nanxiao

共享库

显示共享链接库信息

例子

```
#include <hiredis/hiredis.h>

int main(void)
{
        char a[1026] = {0};
        redisContext *c = NULL;
        void *reply = NULL;

        memset(a, 'a', (sizeof(a) - 1));
        c = redisConnect("127.0.0.1", 6379);
        if (NULL != c)
        {
              reply = redisCommand(c, "set 1 %s", a);
              freeReplyObject(reply);

              reply = redisCommand(c, "get 1");
              freeReplyObject(reply);

              redisFree(c);
        }
        return 0;
}
```

技巧

使用"`info sharedlibrary regex`"命令可以显示程序加载的共享链接库信息，其中`regex`可以是正则表达式，意为显示名字符合`regex`的共享链接库。如果没有`regex`，则列出所有的库。以上面程序为例:

```
(gdb) start
Temporary breakpoint 1 at 0x109f0: file a.c, line 5.
Starting program: /export/home/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 1, main () at a.c:5
5                       char a[1026] = {0};
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0xff3b44a0  0xff3e3490  Yes (*)     /usr/lib/ld.so.1
0xff3325f0  0xff33d4b4  Yes         /usr/local/lib/libhiredis.so.0
0xff3137f0  0xff31a9f4  Yes (*)     /lib/libsocket.so.1
0xff215fd4  0xff28545c  Yes (*)     /lib/libnsl.so.1
0xff0a3a20  0xff14fedc  Yes (*)     /lib/libc.so.1
0xff320400  0xff3234c8  Yes (*)     /platform/SUNW,UltraAX-i2/lib/l
(*): Shared library is missing debugging information.
```

可以看到列出所有加载的共享链接库信息，带“`*`”表示库缺少调试信息。

另外也可以使用正则表达式：

```
(gdb) i sharedlibrary hiredi*
From        To          Syms Read   Shared Object Library
0xff3325f0  0xff33d4b4  Yes         /usr/local/lib/libhiredis.so.0
```

可以看到只列出了一个库信息。
参见gdb手册.

贡献者

nanxiao

脚本

配置gdb init文件

技巧

当gdb启动时，会读取HOME目录和当前目录下的的配置文件，执行里面的命令。这个文件通常为“.gdbinit”。

这里给出了本文档中介绍过的，可以放在“.gdbinit”中的一些配置：

```
# 打印STL容器中的内容
python
import sys
sys.path.insert(0, "/home/xmj/project/gcc-trunk/libstdc++-v3/python
from libstdcxx.v6.printers import register_libstdcxx_printers
register_libstdcxx_printers (None)
end

# 保存历史命令
set history filename ~/.gdb_history
set history save on

# 退出时不显示提示信息
set confirm off

# 按照派生类型打印对象
set print object on

# 打印数组的索引下标
set print array-indexes on

# 每行打印一个结构体成员
set print pretty on
```

欢迎补充。

贡献者

xmj

按何种方式解析脚本文件

例子

```
#include <stdio.h>

typedef struct
{
        int a;
        int b;
        int c;
        int d;
}ex_st;

int main(void) {
        ex_st st = {1, 2, 3, 4};
        printf("%d,%d,%d,%d\n", st.a, st.b, st.c, st.d);
        return 0;
}
```

技巧

gdb支持的脚本文件分为两种：一种是只包含gdb自身命令的脚本，例如“.gdbinit”文件，当gdb在启动时，就会执行“.gdbinit”文件中的命令；此外，gdb还支持其它一些语言写的脚本文件（比如python）。
gdb用“ `set script-extension` ”命令来决定按何种格式来解析脚本文件。它可以取3个值：
a） `off` ：所有的脚本文件都解析成gdb的命令脚本；
b） `soft` ：根据脚本文件扩展名决定如何解析脚本。如果gdb支持解析这种脚本语言（比如python），就按这种语言解析，否则就按命令脚本解析；
c） `strict` ：根据脚本文件扩展名决定如何解析脚本。如果gdb支持解析这种脚本语言（比如python），就按这种语言解析，否则不解析；
以上面程序为例，进行调试：

```
(gdb) start
Temporary breakpoint 1 at 0x4004cd: file a.c, line 12.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:12
12              ex_st st = {1, 2, 3, 4};
(gdb) q
A debugging session is active.

        Inferior 1 [process 24249] will be killed.

Quit anyway? (y or n) y
```

可以看到gdb退出时，默认行为会提示用户是否退出。

下面写一个脚本文件（gdb.py），但内容是一个gdb命令，不是真正的python脚本。用途是退出gdb时不提示：

```
set confirm off
```

再次开始调试：

```
(gdb) start
Temporary breakpoint 1 at 0x4004cd: file a.c, line 12.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:12
12              ex_st st = {1, 2, 3, 4};
(gdb) show script-extension
Script filename extension recognition is "soft".
(gdb) source gdb.py
  File "gdb.py", line 1
    set confirm off
              ^
SyntaxError: invalid syntax
```

可以看到“`script-extension`”默认值是`soft`，接下来执行“`source gdb.py`”,会按照pyhton语言解析gdb.py文件，但是由于这个文件实质上是一个gdb命令脚本，所以解析出错。
再执行一次：

```
(gdb) start
Temporary breakpoint 1 at 0x4004cd: file a.c, line 12.
Starting program: /data2/home/nanxiao/a

Temporary breakpoint 1, main () at a.c:12
12              ex_st st = {1, 2, 3, 4};
(gdb) set script-extension off
(gdb) source gdb.py
(gdb) q
[root@linux:~]$
```

这次把“`script-extension`”值改为`off`，所以脚本会按gdb命令脚本去解析，可以看到这次脚本命令生效了。

参见[gdb手册](#)

贡献者

nanxiao

保存历史命令

技巧

在gdb中，缺省是不保存历史命令的。你可以通过如下命令来设置成保存历史命令：

```
(gdb) set history save on
```

但是，历史命令是缺省保存在了当前目录下的.gdb_history文件中。可以通过如下命令来设置要保存的文件名和路径：

```
(gdb) set history filename fname
```

现在，我们把这两个命令放到$HOME/.gdbinit文件中：

```
set history filename ~/.gdb_history
set history save on
```

好了，下次启动gdb时，你就可以直接查找使用之前的历史命令了。

详情参见gdb手册

贡献者

xmj

源文件

源文件

设置源文件查找路径

例子

```
#include <stdio.h>
#include <time.h>

int main(void) {
        time_t now = time(NULL);
        struct tm local = {0};
        struct tm gmt = {0};

        localtime_r(&now, &local);
        gmtime_r(&now, &gmt);

        return 0;
}
```

技巧

有时gdb不能准确地定位到源文件的位置（比如文件被移走了，等等），此时可以用 `directory` 命令设置查找源文件的路径。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x400560: file a.c, line 5.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:5
5       a.c: No such file or directory.
(gdb) directory ../ki/
Source directories searched: /home/nan/../ki:$cdir:$cwd
(gdb) n
6               struct tm local = {0};
(gdb)
7               struct tm gmt = {0};
(gdb)
9               localtime_r(&now, &local);
(gdb)
10              gmtime_r(&now, &gmt);
(gdb) q
```

可以看到，使用 `directory` （或 `dir`)命令设置源文件的查找目录后，gdb就可以正常地解析源代码了。

如果希望在gdb启动时，加载code的位置，避免每次在gdb中再次输入命令，可以使用gdb的 `-d` 参数

```
gdb -q a.out -d /search/code/some
```

参见gdb手册.

贡献者

nanxiao

替换查找源文件的目录

例子

```
#include <stdio.h>
#include <time.h>

int main(void) {
        time_t now = time(NULL);
        struct tm local = {0};
        struct tm gmt = {0};

        localtime_r(&now, &local);
        gmtime_r(&now, &gmt);

        return 0;
}
```

技巧

有时调试程序时，源代码文件可能已经移到其它的文件夹了。此时可以用`set substitute-path from to`命令设置新的文件夹（`to`）目录替换旧的（`from`）。以上面程序为例：

```
(gdb) start
Temporary breakpoint 1 at 0x400560: file a.c, line 5.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:5
5       a.c: No such file or directory.
(gdb) set substitute-path /home/nan /home/ki
(gdb) n
6                       struct tm local = {0};
(gdb)
7                       struct tm gmt = {0};
(gdb)
9                       localtime_r(&now, &local);
(gdb)
10                      gmtime_r(&now, &gmt);
(gdb)
12                      return 0;
```

调试时，因为源文件已经移到 `/home/ki` 这个文件夹下了，所以gdb找不到源文件。使用 `set substitute-path /home/nan /home/ki` 命令设置源文件的查找目录后，gdb就可以正常地解析源代码了。
参见gdb手册.

贡献者

nanxiao

图形化界面

进入和退出图形化调试界面

例子

```
#include <stdio.h>

void fun1(void)
{
        int i = 0;

        i++;
        i = i * 2;
        printf("%d\n", i);
}

void fun2(void)
{
        int j = 0;

        fun1();
        j++;
        j = j * 2;
        printf("%d\n", j);
}

int main(void)
{
        fun2();
        return 0;
}
```

技巧

启动gdb时指定“`-tui`”参数（例如：`gdb -tui program`），或者运行gdb过程中使用“`Crtl+X+A`”组合键，都可以进入图形化调试界面。以调试上面程序为例：

```
    ┌──a.c─────────────────────────────────────────────────────────────
    │17                 j++;
    │18                 j = j * 2;
    │19                 printf("%d\n", j);
    │20         }
    │21
    │22         int main(void)
    │23         {
B+> │24                 fun2();
    │25                 return 0;
    │26         }
    │27
    │28
    │29
    │30
    │31
    │32
    └──────────────────────────────────────────────────────────────────
native process 22141 In: main
Type "apropos word" to search for commands related to "word"...
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
(gdb)
```

可以看到，显示了当前的程序的进程号，将要执行的代码行号，`PC`寄存器的值。
退出图形化调试界面也是用“`Crtl+X+A`”组合键。
参见gdb手册.

贡献者

nanxiao

显示汇编代码窗口

例子

```
#include <stdio.h>

void fun1(void)
{
        int i = 0;

        i++;
        i = i * 2;
        printf("%d\n", i);
}

void fun2(void)
{
        int j = 0;

        fun1();
        j++;
        j = j * 2;
        printf("%d\n", j);
}

int main(void)
{
        fun2();
        return 0;
}
```

技巧

使用gdb图形化调试界面时，可以使用“ `layout asm` ”命令显示汇编代码窗口。以调试上面程序为例：

```
        ┌──────────────────────────────────────────────────────────────
   >│0x40052b <main+4>                callq  0x4004f3 <fun2>
    │0x400530 <main+9>                mov    $0x0,%eax
    │0x400535 <main+14>               leaveq
    │0x400536 <main+15>               retq
    │0x400537                         nop
    │0x400538                         nop
    │0x400539                         nop
    │0x40053a                         nop
    │0x40053b                         nop
    │0x40053c                         nop
    │0x40053d                         nop
    │0x40053e                         nop
    │0x40053f                         nop
    │0x400540 <__libc_csu_fini>       repz retq
    │0x400542                         data16 data16 data16 data16 nop
    │0x400550 <__libc_csu_init>       mov    %rbp,-0x28(%rsp)
    └──────────────────────────────────────────────────────────────────
native process 44658 In: main

(gdb) start
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
(gdb)
```

可以看到，显示了当前的程序的汇编代码。
如果既想显示源代码，又想显示汇编代码，可以使用“`layout split`”命令：

```
   ┌──a.c─────────────────────────────────────────────────────────────
  >│24              fun2();
   │25              return 0;
   │26      }
   │27
   │28
   │29
   │30
   └──────────────────────────────────────────────────────────────────
  >│0x40052b <main+4>       callq  0x4004f3 <fun2>
   │0x400530 <main+9>       mov    $0x0,%eax
   │0x400535 <main+14>      leaveq
   │0x400536 <main+15>      retq
   │0x400537                nop
   │0x400538                nop
   │0x400539                nop
   │0x40053a                nop
   └──────────────────────────────────────────────────────────────────
native process 44658 In: main

(gdb) start
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
(gdb)
```

可以看到上面显示的是源代码，下面显示的是汇编代码。
参见gdb手册.

贡献者

nanxiao

显示寄存器窗口

例子

```
#include <stdio.h>

void fun1(void)
{
        int i = 0;

        i++;
        i = i * 2;
        printf("%d\n", i);
}

void fun2(void)
{
        int j = 0;

        fun1();
        j++;
        j = j * 2;
        printf("%d\n", j);
}

int main(void)
{
        fun2();
        return 0;
}
```

技巧

使用gdb图形化调试界面时，可以使用“`layout regs`”命令显示寄存器窗口。以调试上面程序为例：

```
┌──Register group: general──────────────────────────────────────────
│rax            0x34e4590f60     227169341280     rbx            0x
│rcx            0x0      0                        rdx            0x
│rsi            0x7fffffffe4a8   140737488348328  rdi            0x
│rbp            0x7fffffffe3c0   0x7fffffffe3c0   rsp            0x
│r8             0x34e458f300     227169334016     r9             0x
│r10            0x7fffffffe210   140737488347664  r11            0x
│r12            0x4003e0 4195296                 r13            0x
└───────────────────────────────────────────────────────────────────
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21
   │22      int main(void)
   │23      {
  >│24              fun2();
   └────────────────────────────────────────────────────────────────
native process 12552 In: main
Reading symbols from a...done.
(gdb) start
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
(gdb)
```

可以看到，显示了通用寄存器的内容。
如果想查看浮点寄存器，可以使用“`tui reg float`”命令：

```
┌──Register group: float──────────────────────────────────────────────
│st0            0        (raw 0x00000000000000000000)
│st1            0        (raw 0x00000000000000000000)
│st2            0        (raw 0x00000000000000000000)
│st3            0        (raw 0x00000000000000000000)
│st4            0        (raw 0x00000000000000000000)
│st5            0        (raw 0x00000000000000000000)
│st6            0        (raw 0x00000000000000000000)
└─────────────────────────────────────────────────────────────────────
   │16              fun1();
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21
   │22      int main(void)
   │23      {
   └──────────────────────────────────────────────────────────────────
native process 12552 In: main
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
(gdb) tui reg float
```

“`tui reg system`”命令显示系统寄存器：

```
┌──Register group: system─────────────────────────────────────────
│orig_rax       0xffffffffffffffff       -1
│
│
│
│
│
│
│
└─────────────────────────────────────────────────────────────────
   │16              fun1();
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21
   │22      int main(void)
   │23      {
   └──────────────────────────────────────────────────────────────
native process 12552 In: main

Temporary breakpoint 1, main () at a.c:24
(gdb) tui reg system
(gdb)
```

想切换回显示通用寄存器内容，可以使用“ `tui reg general` ”命令：

```
┌──Register group: general────────────────────────────────────────
│rax            0x34e4590f60     227169341280     rbx            0x
│rcx            0x0      0                        rdx            0x
│rsi            0x7fffffffe4a8   140737488348328  rdi            0x
│rbp            0x7fffffffe3c0   0x7fffffffe3c0   rsp            0x
│r8             0x34e458f300     227169334016     r9             0x
│r10            0x7fffffffe210   140737488347664  r11            0x
│r12            0x4003e0 4195296                  r13            0x
└─────────────────────────────────────────────────────────────────
   │16              fun1();
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21
   │22      int main(void)
   │23      {
   └──────────────────────────────────────────────────────────────
native process 12552 In: main
(gdb) tui reg general
(gdb)
```

参见gdb手册.

贡献者

nanxiao

调整窗口大小

例子

```
#include <stdio.h>

void fun1(void)
{
        int i = 0;

        i++;
        i = i * 2;
        printf("%d\n", i);
}

void fun2(void)
{
        int j = 0;

        fun1();
        j++;
        j = j * 2;
        printf("%d\n", j);
}

int main(void)
{
        fun2();
        return 0;
}
```

技巧

使用gdb图形化调试界面时，可以使用“`winheight <win_name> [+ | -]count`”命令调整窗口大小（`winheight`缩写为`win`。`win_name`可以是`src`、`cmd`、`asm`和`regs`）。以调试上面程序为例，这是原始的`src`窗口大小：

```
   ┌──a.c──────────────────────────────────────────────────────────────
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21      int main(void)
   │23      {
   │24              fun2();
B+>│25
   │                return 0;
   │26      }
   │27
   │
   │
   │
   │
   │
   └───────────────────────────────────────────────────────────────────
native process 9667 In: main
Usage: winheight <win_name> [+ | -] <#lines>
(gdb) start
Temporary breakpoint 1 at 0x40052b: file a.c, line 24.
Starting program: /home/nan/a

Temporary breakpoint 1, main () at a.c:24
```

执行“`winheight src -5`”命令后：

```
   ┌──a.c──────────────────────────────────────────────────────────────
   │17              j++;
   │18              j = j * 2;
   │19              printf("%d\n", j);
   │20      }
   │21
   │22      int main(void)
   │23      {
  >│24              fun2();
   │25              return 0;
   │26      }
   │27
   └───────────────────────────────────────────────────────────────────
native process 9667 In: main
Usage: winheight <win_name> [+ | -] <#lines>
(gdb)
```

可以看到窗口变小了。
接着执行“`winheight src +5`”命令：

```
   ┌──a.c─────────────────────────────────────────────────────────
   │17                  j++;
   │18                  j = j * 2;
   │19                  printf("%d\n", j);
   │20          }
   │21
   │22          int main(void)
   │23          {
  >│24                  fun2();
   │25                  return 0;
   │26          }
   │27
   │28
   │29
   │30
   │31
   │32
   └──────────────────────────────────────────────────────────────
native process 9667 In: main
Usage: winheight <win_name> [+ | -] <#lines>
(gdb)
```

可以看到窗口恢复了原样。
参见gdb手册.

贡献者

nanxiao

其它

命令行选项的格式

技巧

gdb的帮助信息和在线文档对于长选项的形式使用了不同的风格。你可能有点迷惑，gdb的长选项究竟应该是“-”，还是“--”？

是的，这两种方式都可以。例如：

```
$ gdb -help
$ gdb --help

$ gdb -args ./a.out a b c
$ gdb --args ./a.out a b c
```

好吧，使用短的。

贡献者

xmj

支持预处理器宏信息

例子

```
#include <stdio.h>

#define NAME "Joe"

int main()
{
  printf ("Hello %s\n", NAME);
  return 0;
}
```

技巧

使用 `gcc -g` 编译生成的程序，是不包含预处理器宏信息的：

```
(gdb) p NAME
No symbol "NAME" in current context.
```

如果想在gdb中查看宏信息，可以使用 `gcc -g3` 进行编译：

```
(gdb) p NAME
$1 = "Joe"
```

关于预处理器宏的命令，参见gdb手册

贡献者

xmj

使用命令的缩写形式

技巧

在gdb中，你不用必须输入完整的命令，只需命令的（前）几个字母即可。规则是，只要这个缩写不会和其它命令有歧义（注，是否有歧义，这个规则从文档上看不出，看起来需要查看gdb的源代码，或者在实际使用中进行总结）。也可以使用tab键进行命令补全。

其中许多常用命令只使用第一个字母就可以，比如：

```
b -> break
c -> continue
d -> delete
f -> frame
i -> info
j -> jump
l -> list
n -> next
p -> print
r -> run
s -> step
u -> until
```

也有使用两个或几个字母的，比如：

```
aw -> awatch
bt -> backtrace
dir -> directory
disas -> disassemble
fin -> finish
ig -> ignore
ni -> nexti
rw -> rwatch
si -> stepi
tb -> tbreak
wa -> watch
win -> winheight
```

另外，如果直接按回车键，会重复执行上一次的命令。

贡献者

xmj

nanxiao

在gdb中执行shell命令和make

技巧

你可以不离开gdb，直接执行shell命令，比如：

```
(gdb) shell ls
```

或

```
(gdb) !ls
```

这里，"!"和命令之间不需要有空格（即，有也成）。

特别是当你在构建环境(build目录)下调试程序的时候，可以直接运行make：

```
(gdb) make CFLAGS="-g -O0"
```

详情参见gdb手册

贡献者

xmj

在gdb中执行cd和pwd命令

技巧

是的，gdb确实支持这两个命令，虽然我没有想到它们有什么特别的用处。

也许，当你启动gdb之后，发现需要切换工作目录，但又不想退出gdb的时候：

```
(gdb) pwd
Working directory /home/xmj.
(gdb) cd tmp
Working directory /home/xmj/tmp.
```

详情参见gdb手册

贡献者

xmj

设置命令提示符

例子

```
$ gdb -q `which gdb`
Reading symbols from /home/xmj/install/binutils-gdb-git/bin/gdb...d
(gdb) r -q
Starting program: /home/xmj/install/binutils-gdb-git/bin/gdb -q
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db
(gdb)
```

技巧

当你用gdb来调试gdb的时候，通过设置命令提示符，可以帮助你区分这两个gdb：

```
$ gdb -q `which gdb`
Reading symbols from /home/xmj/install/binutils-gdb-git/bin/gdb...d
(gdb) set prompt (main gdb)
(main gdb) r -q
Starting program: /home/xmj/install/binutils-gdb-git/bin/gdb -q
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db
(gdb)
```

注意，这里 `set prompt (main gdb)` 结尾处是有一个空格的。

详情参见gdb手册

贡献者

xmj

设置被调试程序的参数

技巧

可以在gdb启动时，通过选项指定被调试程序的参数，例如：

```
$ gdb -args ./a.out a b c
```

也可以在gdb中，通过命令来设置，例如：

```
(gdb) set args a b c
(gdb) show args
Argument list to give program being debugged when it is started is
```

也可以在运行程序时，直接指定：

```
(gdb) r a b
Starting program: /home/xmj/tmp/a.out a b
(gdb) show args
Argument list to give program being debugged when it is started is
(gdb) r
Starting program: /home/xmj/tmp/a.out a b
```

可以看出，参数已经被保存了，下次运行时直接运行 `run` 命令，即可。

有意的是，如果我接下来，想让参数为空，该怎么办？是的，直接：

```
(gdb) set args
```

详情参见gdb手册

贡献者

xmj

设置被调试程序的环境变量

例子

```
(gdb) u 309
Warning: couldn't activate thread debugging using libthread_db: Car
Warning: couldn't activate thread debugging using libthread_db: Car
warning: Unable to find libthread_db matching inferior's thread lib
```

技巧

在gdb中，可以通过命令`set env varname=value`来设置被调试程序的环境变量。对于上面的例子，网上可以搜到一些解决方法，其中一种方法就是设置LD_PRELOAD环境变量：

```
set env LD_PRELOAD=/lib/x86_64-linux-gnu/libpthread.so.0
```

注意，这个实际路径在不同的机器环境下可能不一样。把这个命令加到~/.gdbinit文件中，就可以了。

详情参见gdb手册

贡献者

xmj

得到命令的帮助信息

技巧

使用`help`命令可以得到gdb的命令帮助信息：

（1）`help`命令不加任何参数会得到命令的分类：

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the pr
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

（2）当输入`help class`命令时，可以得到这个类别下所有命令的列表和命令功能：

```
(gdb) help data
Examining data.

List of commands:

append -- Append target code/data to a local file
append binary -- Append target code/data to a raw binary file
append binary memory -- Append contents of memory to a raw binary f
append binary value -- Append the value of an expression to a raw b
append memory -- Append contents of memory to a raw binary file
append value -- Append the value of an expression to a raw binary f
call -- Call a function in the program
disassemble -- Disassemble a specified section of memory
display -- Print value of expression EXP each time the program stop
dump -- Dump target code/data to a local file
dump binary -- Write target code/data to a raw binary file
dump binary memory -- Write contents of memory to a raw binary file
dump binary value -- Write the value of an expression to a raw bina
......
```

（3）也可以用`help command`命令得到某一个具体命令的用法：

```
(gdb) help mem
Define attributes for memory region or reset memory region handling
Usage: mem auto
   mem <lo addr> <hi addr> [<mode> <width> <cache>],
where <mode>  may be rw (read/write), ro (read-only) or wo (write-o
  <width> may be 8, 16, 32, or 64, and
  <cache> may be cache or nocache
```

（4）用`apropos regexp`命令查找所有符合`regexp`正则表达式的命令信息：

```
(gdb) apropos set
awatch -- Set a watchpoint for an expression
b -- Set breakpoint at specified line or function
br -- Set breakpoint at specified line or function
bre -- Set breakpoint at specified line or function
brea -- Set breakpoint at specified line or function
......
```

详情参见gdb手册

贡献者

nanxiao

记录执行gdb的过程

例子

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
        char str1[] = "abcd";
        wchar_t str2[] = L"abcd";

        return 0;
}
```

技巧

用gdb调试程序时，可以使用“`set logging on`”命令把执行gdb的过程记录下来，方便以后自己参考或是别人帮忙分析。默认的日志文件是“`gdb.txt`”，也可以用“`set logging file file`”改成别的名字。以上面程序为例：

```
(gdb) set logging file log.txt
(gdb) set logging on
Copying output to log.txt.
(gdb) start
Temporary breakpoint 1 at 0x8050abe: file a.c, line 6.
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 1, main () at a.c:6
6               char str1[] = "abcd";
(gdb) n
7               wchar_t str2[] = L"abcd";
(gdb) x/s str1
0x804779f:      "abcd"
(gdb) n
9               return 0;
(gdb) x/ws str2
0x8047788:      U"abcd"
(gdb) q
A debugging session is active.

        Inferior 1 [process 9931    ] will be killed.

Quit anyway? (y or n) y
```

执行完后，查看log.txt文件：

```
bash-3.2# cat log.txt
Temporary breakpoint 1 at 0x8050abe: file a.c, line 6.
Starting program: /data1/nan/a
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
[Switching to Thread 1 (LWP 1)]

Temporary breakpoint 1, main () at a.c:6
6               char str1[] = "abcd";
7               wchar_t str2[] = L"abcd";
0x804779f:      "abcd"
9               return 0;
0x8047788:      U"abcd"
A debugging session is active.

        Inferior 1 [process 9931    ] will be killed.

Quit anyway? (y or n)
```

可以看到log.txt详细地记录了gdb的执行过程。

此外“ `set logging overwrite on` ”命令可以让输出覆盖之前的日志文件；而“ `set logging redirect on` ”命令会让gdb的日志不会打印在终端。
参见gdb手册.

贡献者

nanxiao